Three-Valued Abstraction and Heuristic-Guided Refinement for Verifying Concurrent Systems

Dissertation
in Computer Science

submitted to the
Faculty of Electrical Engineering,
Computer Science and Mathematics
University of Paderborn

in partial fulfillment of the requirements for the degree of
doctor rerum naturalium

Nils Timm

Paderborn, May 2013

Abstract

Software systems are playing an increasing role in our everyday life, and as the amount of software applications grows, so does their complexity and the relevance of their computations. Software components can be found in many systems that are charged with safety-critical tasks, such as control systems for aviation or power plants. Hence, software verification techniques that are capable of proving the absence of critical errors are becoming more and more important in the field software engineering.

A well-established approach to software verification is *model checking*. Applying this technique involves an exhaustive exploration of a state space model corresponding to the system under consideration. The major challenge in model checking is the so-called *state explosion problem*: The state space of a software system grows exponentially with its size. Thus, the straightforward modelling of real-life systems practically impossible. A common approach to this problem is the application of *abstraction techniques*, which reduce the original state space by mapping it on a significantly smaller abstract one. Abstraction inherently involves a loss of information, and thus, the resulting abstract model may be too imprecise for a definite result in verification. Therefore, abstraction is typically combined with *abstraction refinement*: An initially very coarse abstract model is iteratively refined, i.e. enriched with new details about the original system, until a level of abstraction is reached that is precise enough for a definite outcome. Abstraction refinement-based model checking is fully automatable and it is considered as one of the most promising approaches to the state explosion problem in verification. However, it is still faced with a number of challenges. There exist several types of abstraction techniques and not every type is equally well-suited for all kinds of systems and verification tasks. Moreover, the selection of adequate refinement steps is nontrivial and typically the most crucial part of the overall approach: Unfavourable refinement decisions can compromise the state space-reducing effect of abstraction, and as a consequence, can easily lead to the failure of verification. It is, however, hard to predict which refinement steps will eventually be expedient for verification – and which not.

In this thesis, we approach the previously addressed challenges of abstraction refinement-based model checking by focusing on one specific type of software system: *Concurrent systems* are compositions of interleaved executed software processes that communicate via shared variables or message passing, which makes their verification particularly difficult. However, these systems also reveal a high amount of structural information – in particular, the communication dependencies between processes – that we exploit for improving abstraction and refinement. To this end, we introduce a comprehensive verification framework for concurrent systems.

Our approach to abstraction is based on a combination of *predicate abstraction* – a data abstraction technique that replaces concrete system variables by predicates – and *spotlight abstraction* – a technique that abstracts away entire processes of a concurrent system. We thus tackle the two major causes of state explosion for concurrent systems. Another key feature of our approach is the use of a *three-valued* abstract domain. Properties in our models can take the values *true*, *false* and *unknown*, which enables us to explicitly model the loss of information due to abstraction: All *true* and *false* results obtained via model checking can be transferred to the original system, only an *unknown* outcome necessitates abstraction refinement.

For automatically refining the abstract models we follow the concept of *counterexample-guided abstraction refinement* (CEGAR). Counterexamples are '*unknown*' error paths in the abstract model that typically hint at *several* possible ways to resolve the uncertainty via refinement. In our scenario, these refinement steps can involve the addition of new predicates or new processes. However, not every potential refinement step is expedient, which makes the selection of an appropriate step an exceedingly difficult task. Therefore, we introduce a variant of CEGAR enhanced by *heuristic guidance*: Based on an iterative *abstraction dependence analysis* the possible refinement steps are heuristically evaluated with regard to their benefit for the current verification task, and the best evaluated step is chosen for refinement. In two case studies, we demonstrate that our heuristic approach can significantly improve the performance of abstraction-refinement-based verification of concurrent systems.

Our developed verification framework primarily allows for reasoning about safety and liveness properties of concurrent systems that are composed of a fixed number of processes. However, we also introduce an extension that facilitates the verification of *parameterised systems* – compositions of an unbounded number of homogeneous processes. Our extension is based on a combination spotlight abstraction with *symmetry reduction*, a reduction technique that exploits the homogeneity in parameterised systems.

Zusammenfassung

Softwaresysteme nehmen eine immer bedeutendere Rolle in unserem Alltag ein. Mit der steigenden Zahl von Anwendungsbereichen von Software steigt auch ihre Komplexität und die Relevanz ihrer Berechnungen. Softwarekomponenten finden sich in unzähligen Systemen die mit sicherheitskritischen Aufgaben betraut sind, beispielsweise Systeme zur Regelung des Flugverkehrs oder zur Kontrolle von Kraftwerken. Im Bereich des Software Engineering werden daher formale Verifikationstechniken, Methoden zum Nachweis der Korrektheit von Software, immer wichtiger.

Ein etabliertes Verifikationsverfahren ist das *Model Checking*. Die Anwendung dieser Technik geht einher mit einer vollständigen Exploration eines Zustandsraummodells korrespondierend zum untersuchten System. Die wesentliche Herausforderung ist hierbei das sogenannte *State Explosion Problem*: Der Zustandsraum eines Softwaresystems wächst exponentiell mit der Systemgröße. Für große Systeme ist eine direkte Modellierung praktisch nicht durchführbar. Ein verbreiteter Ansatz ist daher die Anwendung von *Abstraktionstechniken*, welche den tatsächlichen Zustandsraum auf einen signifikant kleineren abstrakten Zustandsraum reduzieren. Abstraktion geht grundsätzlich einher mit dem Verlust von Information. Deshalb ist es möglich, dass die erzeugten abstrakten Modelle zu ungenau für ein definitives Ergebnis bei der Verifikation sind. Abstraktion wird daher typischerweise mit *Abstraktionsverfeinerung* kombiniert: Ein initial sehr grobes Modell wird so lange verfeinert, d.h. um neue Details über das Originalsystem erweitert, bis ein Abstraktionslevel erreicht wurde, das ein definites Verifikationsergebnis zulässt. Abstraktionsverfeinerungsbasiertes Model Checking ist vollständig automatisierbar und gilt als einer der vielversprechendsten Ansätze zur Bewältigung des State Explosion Problems. Jedoch ist der Einsatz dieses Verfahrens nach wie vor mit großen Herausforderungen verbunden. Es existieren verschiedene Formen der Abstraktion und nicht jede Form ist für alle Arten von Systemen und Verifikationsaufgaben gleich gut geeignet. Zudem stellt die Auswahl geeigneter Verfeinerungsschritte einen nicht trivialen und typischerweise entscheidenden Teil des gesamten Verfahrens dar: Ungünstige Verfeinerungsentscheidungen

können dem zustandsraumreduzierenden Effekt der Abstraktion entgegenwirken und somit zu einem Fehlschlagen der Verifikation führen. Es ist jedoch schwer vorhersagbar, welche Verfeinerungsschritte sich letztendlich als günstig für die Verifikation erweisen – und welche nicht.

Diese Arbeit behandelt die Entwicklung eines Verifikationsverfahrens, welches die zuvor erläuterten Herausforderungen in den Bereichen Abstraktion und Verfeinerung angeht. Hierbei wird der Fokus auf eine spezifische Art von Softwaresystemen gelegt: *Nebenläufige Systeme* sind Kompositionen verzahnt ausgeführter Softwareprozesse, welche über globale Variablen oder Nachrichtenaustausch kommunizieren. Die verzahnte Ausführung führt zu einer hohen Komplexität der Verifikation solcher Systeme. Jedoch geben nebenläufige Systeme leicht erfassbare Strukturinformationen preis – insbesondere bezüglich der Kommunikationsabhängigkeiten zwischen den Prozessen. Diese Informationen werden in dem hier entwickelten Verifikationsverfahren zur Verbesserung von Abstraktion und Verfeinerung genutzt.

Der hierbei eingesetzte Abstraktionsansatz basiert auf einer Kombination von *Prädikatabstraktion* – einer Datenabstraktionstechnik, welche konkrete Systemvariablen durch Prädikate ersetzt – und *Spotlightabstraktion* – einer Technik, welche ganze Prozesse eines nebenläufigen Systems wegabstrahiert. Damit werden zwei der Hauptursachen der Zustandskomplexität von nebenläufigen Systemen angegangen. Eine weitere Besonderheit des entwickelten Verfahrens ist die Verwendung einer dreiwertigen abstrakten Domäne. Eigenschaften in den Zustandsmodellen können die Werte *wahr*, *falsch* und *unbekannt* annehmen, wodurch sich der abstraktionsbedingte Informationsverlust explizit modellieren lässt: Alle *wahr*- und *falsch*-Resultate die sich via Model Checking ergeben, lassen sich auf das Originalsystem übertragen. Lediglich ein *unbekannt*-Resultat erfordert die Verfeinerung der Abstraktion.

Zur automatischen Verfeinerung abstrakter Modelle wird in dem entwickelten Ansatz auf das Konzept der *gegenbeispielgeleiteten Abstraktionsverfeinerung* (englisch *counterexample-guided abstraction refinement* (CEGAR)) zurückgegriffen. Gegenbeispiele entsprechen '*unbekannten*' Fehlerpfaden innerhalb des abstrakten Modells. Diese verweisen auf eine Reihe möglicher Verfeinerungsschritte zur Auflösung der Ungewissheit. Dies können in dem hier entwickelten Verfahren sowohl neue Prädikate als auch neue Prozesse sein. Typischerweise erweist sich jedoch nicht jeder potentielle Verfeinerungsschritt letztendlich als zielführend, was die Auswahl geeigneter Schritte zu einer generell schwierigen Aufgabe macht. In dieser Arbeit wird eine Erweiterung des ursprünglichen CEGAR Konzepts eingeführt, basierend auf *heuristischer Leitung*: Auf Grundlage einer iterativen *Abstraktionsabhängigkeitsanalyse* werden die potentiellen Verfeinerungsschritte bezüglich ihres Nutzens für die aktuelle Verifikationsaufgabe heuristisch bewertet, und der bestbewertete Schritt wird zur Verfeinerung ausgewählt. Anhand von zwei Fallstudien wird gezeigt, dass dieser heuristische Ansatz zu einer signifikanten Leistungssteigerung von abstraktionsverfeinerungsbasierter Verifikation nebenläufiger Systeme beitragen kann.

Das im Rahmen dieser Arbeit entwickelte Verfahren erlaubt primär die Verifikation von Sicherheits- und Lebendigkeitseigenschaften von nebenläufigen Systemen, die aus einer festen Anzahl von Prozessen zusammengesetzt sind. Überdies wird jedoch auch eine Erweiterung vorgestellt, welche die Verifikation *parametrisierter Systeme*, Kompositionen einer unbeschränkten Anzahl von homogenen Prozessen, erlaubt. Diese Erweiterung basiert auf einer Kombination von Spotlightabstraktion und *Symmetriereduktion*, einer Reduktionstechnik zur Ausnutzung der Homogenität in parametrisierten Systemen.

Acknowledgements

Many thanks to all who supported and encouraged me during my PhD studies. My special gratitude goes to Professor Dr. Heike Wehrheim for her constant support and for many valuable discussions and suggestions. I would also like to thank Professor Dr. Hans Kleine Büning who agreed to review my thesis. Moreover, I am grateful to Elisabeth Schlatt and to my Phd colleagues Galina Besova, Tobias Isenberg, Marie-Christine Jakobs, Alexander Schremmer, Dominik Steenken, Oleg Travkin, Sven Walther and Steffen Ziegert for the pleasant atmosphere in our research group. Of course, my thanks also goes to my former colleagues Björn Metzler and Thomas Ruhroth, who supported me a lot in the beginning of my PhD – and I am especially thankful to Daniel Wonisch for his helpful suggestions on my research. Another big thanks goes to Mike Czech for his valuable help with implementing the heuristic framework. Moreover, I would like to thank all my friends, especially my roommates, who encouraged me in so many ways. Last but not least, I want to express my gratitude to my family, in particular to my parents, my grandparents, and my sister. Thank you for your enduring support.

Contents

Chapter 1
Introduction

Software is playing an ever-increasing role in our lives. Everyday we are faced with several devices that rely to a great extent on software technology. Cell phones, electronic payment terminals and automotive trip computers are only some examples of software-based systems that are actively used by billions of people worldwide. Nevertheless, the major part of software is running beyond our immediate awareness. Software is integrated in systems for logistics, medical therapy, aviation control and plenty more. Thus, it can essentially contribute to our daily convenience, health and safety. Moreover, software systems are of great economic importance. A vast number of manufacturing and business processes are in fact controlled by software, and in the future, it is expected that software technology will find its way into many more areas of economy and personal life.

As the amount of software applications grows, so does their complexity and the relevance of their computations. Software is usually part of larger systems composed of several interacting components. Concurrency and distribution are typical characteristics of these integrated systems that are often charged with safety-critical tasks, such as the control of power plants. Moreover, the software development process is commonly subject to very sharp time and cost constraints. Thus, today's software developers are faced with enormous challenges. Highly complex and reliable software has to be developed and integrated into large-scale systems in time. – However, the increasing complexity also involves an increasing number of potential defects. Since it is virtually impossible to avoid all kinds of errors in software development, the application of additional methods for establishing software reliability is indispensable. Cases of bad or insufficient quality management occur frequently. – Two recent examples: The almost crash of an Airbus A330 in 2008, which seriously injured several passengers, was caused by incorrect software. Another sort of fatal crash hit the Wall Street company Knight Capital in 2012. Erroneous trading software caused a loss of hundreds of millions of U.S. dollars to the company, and thus, led to its near bankruptcy.

Software verification is a discipline of engineering that encompasses all methods for detecting defects in software. This discipline is also a field of very active research, since the ever-increasing complexity of today's software demands ever more efficient methods for establishing reliability. – It is a fact that several dramatic software failures, including the two aforementioned ones, could have been prevented by the application of existing verification techniques, and thereby, a lot of harm and costs could have been avoided. In the subsequent section we will provide a brief overview of established methods from the field of software verification.

1.1 Software Verification

Software verification is a collective term for methodical approaches to prove or refute properties of a software system. A fundamental prerequisite for any form of verification is the specification of correctness requirements. Correctness of software is not an absolute property – it is relative to certain previously defined requirements. These requirements are typically qualitative or quantitative assertions over the software, such as the exclusion of specific undesirable behaviour. Verifying a software system generally means to check if a violation of the correctness requirements is detectable.

In the field of software verification we can distinguish two major approaches: testing and applying formal methods. *Software testing* [17] is an integral part of nearly every software development project. It involves the actual execution of the considered software system. The behaviour of the system is examined under different inputs – the so-called test cases. Defects are revealed by comparing the observed behaviour with the previously specified correctness requirements. Testing is generally very efficient and it facilitates the detection of the majority of errors introduced during development. Thus, it can be regarded as an absolute necessity for the success of a software project. However, testing is inherently not exhaustive and not complete. For large-scale software systems it is practically not feasible to consider *all* possible inputs as test cases. In accordance with this fact, Dijkstra [51] stated that

> "program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence."

The incompleteness of software testing is a serious drawback when the absence of certain errors like deadlocks or race conditions is indispensable. – For safety-critical software systems testing alone is certainly not sufficient to ensure the urgently required reliability: Systems for aviation control, driving assistance or medical therapy that are not *guaranteed* to be free of certain bad behaviour are of course not acceptable for practical use. Hence, for such systems testing is usually complemented by the application of formal methods.

Formal methods [40] are specification and verification techniques that rely on mathematical rigour. In contrast to classical software testing, these methods are commonly not based on a *direct* inspection of a software system. The transformation of the system under consideration into a simpler model is an integral part of the application of formal methods. The constructed models are typically *abstract* in the sense that they are restricted to exactly the details that are relevant for the verification of a certain requirement. Another key feature of formal methods is the mathematical accuracy of both the modelling of the system and the specification of the correctness requirements. Established concepts that are employed here are formal languages, automata theory and mathematical logic. Moreover, formal methods provide algorithmic procedures for the systematic exploration of the constructed system models. A prominent example is *model checking* [41, 8] – a fully-automatic and *exhaustive* procedure for model exploration that facilitates the detection of requirement violations *as well as the proof of their absence*. Formal methods thus can overcome the infeasibility of a complete analysis of software systems by the construction of simpler abstract models that allow for an exhaustive exploration. Hence, the application of formal methods for software verification is the ideal complement to testing: Specific safety-critical errors can be excluded based on the employment of model checking, whereas testing provides for the detection of the major part of less serious defects.

The aforementioned *restriction to the relevant details* in modelling is one of the most crucial issues in research on formal methods. Abstracting the system model is absolutely essential for the feasibility of an exhaustive analysis. However, too many or inadequate restrictions may result in a model that does not permit to draw any conclusions about the requirement of interest. The choice of an appropriate level of abstraction is highly nontrivial and may require a lot of manual effort and later corrections. In addition, the increasing complexity of today's software systems makes it even more challenging to construct small but precise system models. Current research in this field thus particularly focuses on the development of techniques for the automatic construction of reduced, or rather *abstract*, models for formal verification.

1.2 Modelling, Abstraction and Refinement

Applying the formal method *model checking* involves an exhaustive exploration of a software system's *state space*. – In a nutshell, the system under consideration is modelled as a state transition graph such that the unreachability of certain error states (or in other cases: the recurrent reachability of certain 'good' states) implies the correctness of the system. Figure 1.1 illustrates the modelling of a single software component as a transition graph. As we can see, the black error state is not reachable from the initial state. Thus, applying model checking would return that the system is correct.

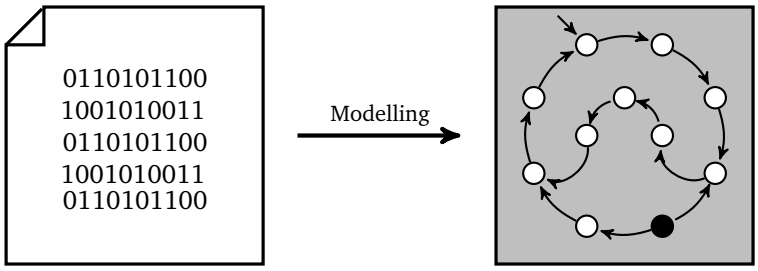

Fig. 1.1 Modelling a software component as a state transition graph. The black state in the transition graph denotes an error state.

The number of states in the transition graph, however, grows *exponentially* with the size of the modelled system, i.e. the number of its variables and components. Thus, large data domains and concurrency – typical characteristics of modern software systems – are among the main causes of the so-called *state explosion problem* in verification that makes a straightforward modelling of real-life systems practically impossible. Many systems even have infinite state spaces, which makes their verification undecidable in general.

A number of approaches have been proposed that address the state explosion problem by constructing reduced state space models for verification: Reduction techniques like *partial order reduction* [64] or *symmetry reduction* [58] exploit specific structural properties of the considered system in order to obtain a smaller state space. *Compositional verification* [122] is an approach that is based on the decomposition of the original verification problem into several less complex subproblems. The field of *abstraction* (e.g. [34, 11, 112]) comprises a wide range of techniques that reduce the state space by hiding certain details, such as concrete data values or entire components of a concurrent system. Abstraction is among the most promising approaches to the state explosion problem. Its application can lead to a substantial decrease of complexity, and besides, it can be combined with other reduction techniques, which facilitates even more efficiency in verification. However, finding the right level of abstraction is exceedingly difficult – abstract models may be either too imprecise for verification, or too complex for an exhaustive exploration. It is thus a common approach to initially construct a very coarse abstract model of the original system, which is then iteratively *refined*, i.e. more details are added, until a level of abstraction is reached that is precise enough for verification. The new details are typically derived from so-called *abstract counterexamples* – error paths in the abstract model that do not necessarily represent feasible behaviour of the original system.

Figuratively speaking, applying abstraction to a system can be regarded as putting a 'grid' on the concrete state space such that the original states are grouped into a small number of abstract states. Refining the abstraction then means to select a finer grid, i.e. to split certain abstract states into more

concrete ones. The schema in Figure 1.2 illustrates the basic principle of abstraction refinement-based verification.

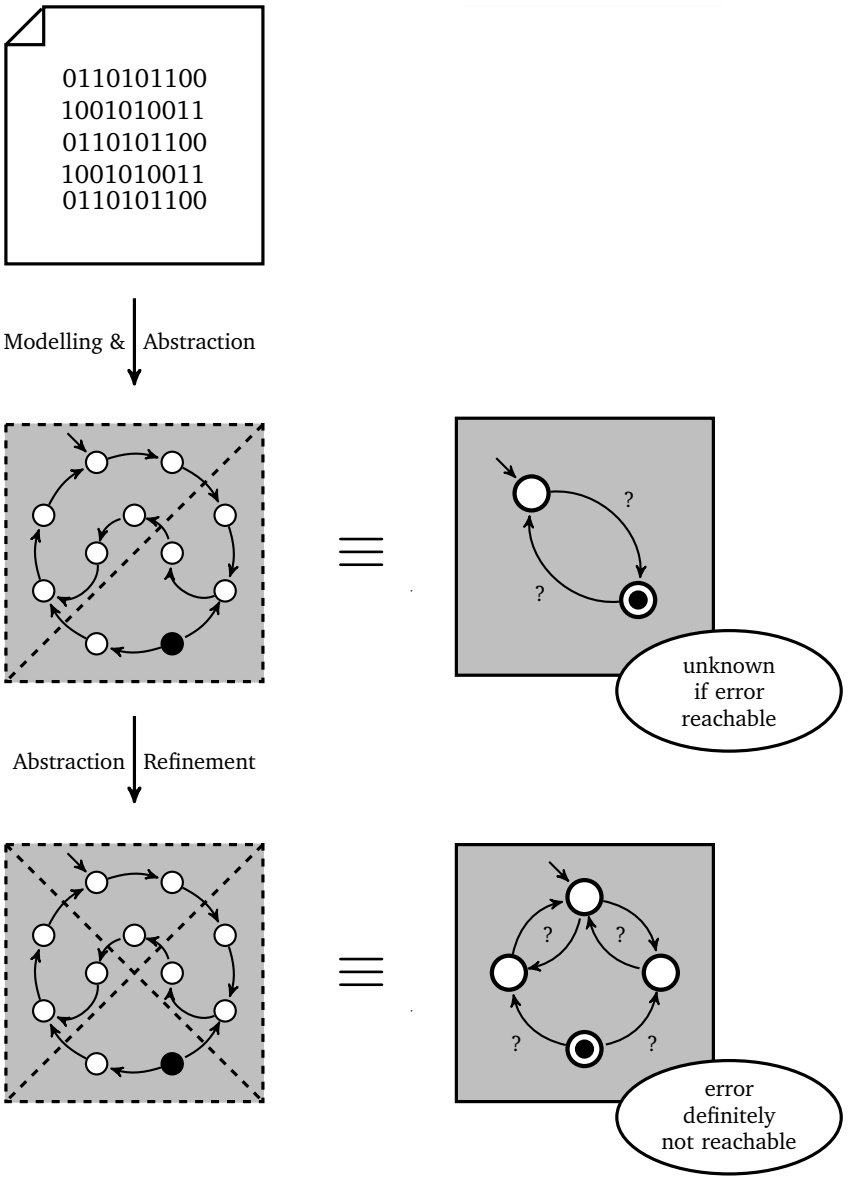

Fig. 1.2 Schematic view of abstraction refinement in formal verification.

Hence, in abstraction refinement-based verification the concrete state space is never explicitly constructed. Instead, the initial abstraction directly groups

the concrete states into a small number of abstract states. In our exemplary schema we have two groups: The upper part of the dashed grid which includes the initial state, and the lower part of the grid which comprises the error state. The resulting abstract transition system (on the right of the schema) is, however, too coarse for a definite result in verification. The ?-tagged transition starting in the initial state is an abstract counterexample that tells us that an error state is *maybe* reachable. In the refinement step, we rule out the abstract counterexample by splitting the two abstract states into four. As we can see, the refined transition system still exhibits uncertainty, but it is actually precise enough to prove that the error state is *definitely not* reachable.

Existing frameworks for abstraction refinement-based verification usually rely on *predicate abstraction* [11]: The concrete variables of the considered system are replaced by a number of boolean predicates over these variables, which commonly yields an over-approximation of the original system. Thus, refinement means to iteratively add new predicates to the abstract model until a definite result in verification can be obtained. Achieving a definite result typically requires a *larger number* of refinement iterations – and not just a single step like in our illustrating schema. Moreover, there usually exist *several* possible directions for refinement, but it is not straightforward to predict which direction will be expedient and which not. Unfavourable refinement steps can easily lead to an unnecessary explosion of the abstract state space. The development of automatic decision procedures for refinement in order to facilitate efficient verification is thus a challenging field of research.

What makes things even more difficult is that today's software systems typically do not consist of only a single component. *Concurrent systems* are composed of a number of software components, also referred to as *software processes*, that are concurrently executed and that communicate with each other via shared variables or message passing. An illustration of a concurrent software system is depicted in the figure below.

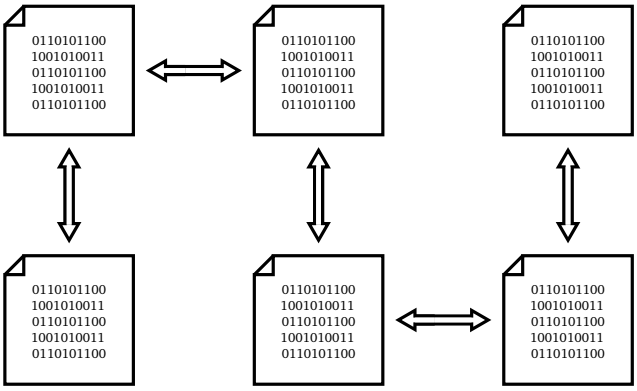

Fig. 1.3 Concurrent system composed of six software processes. The arrows represent potential communication between the processes.

As we have mentioned before, the state space of a system also grows exponentially with the number of its components. Of course, predicate abstraction could be applied to *each* software process of a concurrent system. However, for several verification tasks this would still involve a large and unnecessary overhead. Errors are typically the result of an awkward interplay between a *small* number of processes – and not between *all* the components of the system. Moreover, local correctness requirements of a concurrent system can often be validated based on a small selection of the overall systems components. For the verification of concurrent systems predicate abstraction can be combined with *spotlight abstraction* [123, 112] – a reduction technique that is capable of abstracting away entire processes. The basic principle of spotlight abstraction is illustrated in the figure below.

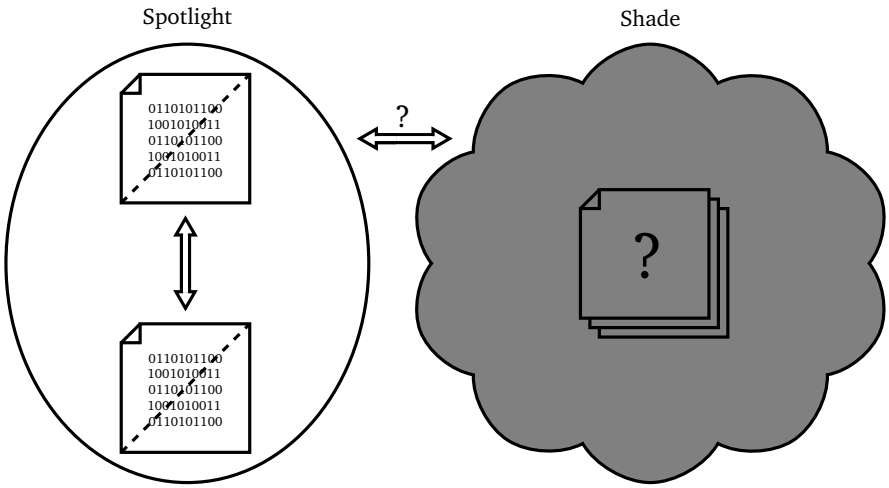

Fig. 1.4 Pictorial representation of spotlight abstraction for concurrent systems.

Hence, applying spotlight abstraction partitions a concurrent system into two parts – a *spotlight* and a *shade*. The processes in the spotlight are typically those that are supposed to be particularly relevant for the verification task, for instance, processes where an error is expected. – The processes of presumably minor importance are initially put into the shade. Now, predicate abstraction is applied to all spotlight processes, whereas the shade processes are collapsed into a single component that coarsely approximates the behaviour of the entire shade. Note that the spotlight and the shade are not independent parts: Due to shared variables or message passing, processes in the shade may affect processes in the spotlight and vice versa. Thus, the inherent loss of information about the shade caused by the summarisation of its processes into a single component may also involve a loss of information about the spotlight processes. In spotlight abstraction this loss of information is characterised

by the introduction of a third truth value *unknown*. Hence, predicates are no longer boolean but *three-valued* and the abstract shade component may set predicates over the variables shared with spotlight processes to *unknown*. This entails that the outcome of spotlight abstraction-based verification can be *unknown* as well, which necessitates refinement. The spotlight principle adds another facet to abstraction refinement. In every refinement iteration either a new predicate can be added to the abstraction or a process can be shifted from the shade to the spotlight. This additional dimension for refinement comes along with enhanced possibilities for constructing the most appropriate, i.e. small *and* precise, models for verification. However, the new scope for refinement decisions also makes it considerably more difficult and challenging to avoid refinement steps that later turn out to be unrewarding. A number of unfavourable refinements can easily result in an abstract model whose size widely exceeds the capabilities of state-of-the-art verification tools.

A common approach to automatically guide the refinement procedure in abstraction-based verification is *counterexample-guided abstraction refinement* (CEGAR) [34]. The basic principle of CEGAR is to first search for an abstract counterexample that does not correspond to feasible behaviour of the concrete system, and then select a refinement step that rules out this counterexample. This is repeated until a definite result in verification can be achieved. Thus, counterexample-guided abstraction refinement can facilitate the detection of reasonable refinements. However, only little research has been spent on advancements of the basic principle of CEGAR. Most existing techniques are based on pure boolean predicate abstraction – neither on a three-valued domain nor on spotlight abstraction – and they are typically restricted to the verification of simple reachability (alias safety) properties. Moreover, CEGAR is a very generic approach to abstraction refinement. Its guidance solely relies on the generated counterexamples. In particular, CEGAR does *not* exploit any further structural information about the system under consideration for its refinement decisions. In many cases, this can even lead to misguidance such that verification fails due to one-sided refinement decisions.

1.3 Contributions of this Work

In this thesis, we approach the previously addressed challenges of abstraction refinement-based verification by focusing on *one specific type* of software system: *Concurrent systems* are in widespread practical use, but the concept of concurrency makes their verification particularly difficult. Nevertheless, these systems also reveal a high amount of easily accessible and useful information – in particular, their communication structure and the resulting dependencies between processes – that we want to maximally exploit for enhancing abstraction and refinement. To this end, we develop a comprehensive framework for

the automatic verification of safety and liveness requirements of concurrent systems. Our specific contributions are as follows.

We base our overall approach on a combination of three-valued predicate abstraction and spotlight abstraction, and thus, tackle both major causes of state explosion, large-domain variables and concurrency. While former approaches to spotlight abstraction [123, 112] abstract away (i.e. lose) the *entire* behaviour of the shade processes, we introduce two extensions of the original spotlight principle that allow us to preserve more concrete behaviour without a significant increase of complexity: *Shade clustering* is a technique for partitioning the shade into clusters of behaviourally similar processes. In *region summarisation* we exploit that summarising certain parts of a process into regions can even remove some uncertainty from the corresponding abstract model. For both extensions we utilise structural information that can be straightforwardly derived from the underlying concurrent system. We prove the soundness of our extensions of spotlight abstraction. That is, we show that all definite behaviour in a resulting abstract model corresponds to feasible behaviour of the original system. – In short, with our contributions in the field of abstraction we want to demonstrate that exploiting additional information about the system under consideration can help to preserve more definite behaviour under abstraction, and thus, to make abstraction-based verification of concurrent systems more efficient.

Our approach to refining imprecise abstractions is an extension of the classical counterexample-guided abstraction refinement. A counterexample typically hints at *several* potential refinement steps – so-called *refinement candidates*. Due to the state explosion problem it is generally not advisable to select *all* candidates for refinement. Most existing CEGAR techniques are based on the detection of minimal refinements that *eliminate* an abstract counterexample. However, in comparison to these classic techniques, we do not employ pure boolean predicate abstraction, but three-valued predicate abstraction together with spotlight abstraction. In our three-valued scenario abstract counterexamples can not only be possibly eliminated, but also *confirmed* via refinement. Moreover, our refinement candidates can be new predicates as well as processes from the shade. These differences generally demand a new approach to refinement. Besides, we again want to exploit structural information about the considered system in order to enhance the refinement procedure. Therefore, we introduce a *heuristic* framework for refinement. We generate *multiple counterexamples* in each iteration, which on the one hand gives us a larger set of refinement candidates, and on the other hand enables us to concretise more than one abstract counterexample in each refinement step. Our heuristic framework relies on an iterative *abstraction dependence analysis*: Based on an analysis of the dependence structure of the underlying concurrent system, the derived refinement candidates are heuristically evaluated with regard to their *benefit* for the current verification task. In each refinement iteration, the *best evaluated* candidate – a predicate or a process – is added to the abstraction.

This enables to guide the refinement procedure in expedient directions and to avoid unnecessary steps. Hence, with our heuristic approach we want to show that refinement decisions in CEGAR – and thus also the performance of the overall verification procedure – can be significantly improved by the untilisation of easily accessible information about the verified system.

Furthermore, we demonstrate that our abstraction refinement-based verification technique can be combined with *symmetry reduction* [58] – which permits us to apply our approach in an even broader context. Verification via the original spotlight principle [112] is limited to reasoning about concurrent systems composed of a *fixed* number of processes. However, for many real-life systems such as network protocols the number of processes is not bounded. Such *parameterised systems* typically consist of an *arbitrary* number of processes that can be divided into a finite number of classes of homogeneous processes. Verifying parameterised systems is undecidable in general, but symmetry reduction techniques can be used to exploit the homogeneity in these systems. In certain cases, this allows to map the infinite state spaces of these unbounded systems to finite abstractions that can be verified. We show that, based on symmetry arguments, our abstraction refinement framework can be extended to the verification of parameterised systems and we prove the soundness of this combined approach. Moreover, we demonstrate that parameterised verification can actually profit from the state space-reducing character of spotlight abstraction.

Finally, we present the implementation of our developed verification framework: We introduce our fully automatic verification tool 3Spot, which takes concurrent systems in a C-like syntax as input. In an experimental evaluation, we investigate how our *heuristic* approach to abstraction refinement can contribute to more efficiency in the verification of concurrent systems.

1.4 Outline

This thesis is structured as follows.

Chapter 2 provides a fundamental introduction to model checking – the formal method that serves as the basis of our approach. We start with introducing classical (boolean) temporal logic model checking, along with all related definitions and terms. In particular, we present Kripke structures as formal models for software systems and we introduce the temporal logic CTL that we employ for the formalisation of correctness requirements. Furthermore, we present the three-valued generalisation of classical model checking. This also involves the introduction of the third truth value *unknown*, which is a key element in our approach. The chapter concludes with an overview of model checking algorithms.

In Chapter 3, we give an introduction to the field of concurrent systems, which includes a definition of the syntax and semantics of such systems as well as a description of the different concepts of communication. We describe the different kinds of concurrent systems that are in the focus of our approach to verification by means of several examples. Moreover, we show how concurrent systems can be represented in a formal way and finally be transferred into a state space model for verification. We conclude this chapter with an overview of typical correctness requirements of concurrent systems that we want to verify with our developed framework.

The Chapters 4, 5 and 6 are the main chapters of this work. In Chapter 4, we present our approach to abstraction. We start with a comprehensive description of the state space reduction techniques predicate abstraction and spotlight abstraction. We show how these two techniques can be effectively combined for the verification of concurrent systems. Furthermore, we introduce two extensions of spotlight abstraction that we have developed within this work: shade clustering and region summarisation. We prove the soundness of our extensions and we demonstrate how spotlight abstraction-based verification can profit from our enhancements. This chapter concludes with a discussion of related work on abstraction in formal verification.

Chapter 5 presents our approach to refinement. starts with an introduction counterexample-guided abstraction refinement (CEGAR). Based on a running example, we illustrate that the efficiency of CEGAR-based verification crucially depends on the choice of adequate refinement steps, which motivates our *heuristic* approach to refinement. Furthermore, we show that for our three-valued abstractions multiple counterexamples can be efficiently generated, which gives us a broader basis for refinement decisions. Finally, we introduce our heuristic framework for abstraction refinement. Based on a structural analysis of the underlying concurrent system we heuristically evaluate the benefit of potential refinement steps, which enables us to guide the refinement in expedient directions, and thus to obtain definite verification results on very small abstractions. A discussion of related work on abstraction refinement concludes this chapter.

In Chapter 6, we show that our approach can be extended to the verification of parameterised systems. We first provide an introduction to parameterised verification and symmetry reduction. Subsequently, we show that symmetry reduction can be effectively integrated into our framework for abstraction refinement. This also encompasses a comprehensive correctness proof of the integration. We furthermore demonstrate that the combination of spotlight abstraction with symmetry arguments enables us to extend our heuristic approach towards the efficient verification of parameterised systems. The chapter is completed with a discussion of related work on parameterised verification.

Chapter 7 describes the implementation of our verification framework. We provide an overview of the essential components of our developed verification tool 3Spot. Moreover, we present two case studies that demonstrate the

applicability of our heuristic approach for larger-scale concurrent systems and we discuss the achieved results.

In the final chapter we conclude this thesis with a summary and we propose promising directions for future research.

Chapter 2
Formal Verification via Model Checking

Formal verification is based on an *exhaustive* analysis of the system under consideration. It not only facilitates the detection of errors, but also the *proof of their absence*. However, a number of challenges are associated with the formal verification of software systems. Research in this field aims to reduce the effort for verification, and thus, to increase its efficiency. We can distinguish different kinds of verification efforts. Naturally, there is a demand for *computational resources*. The construction and exploration of the state space of a software system generally requires a high amount of computation time and memory space. Furthermore, *human resources* may be involved in a verification procedure. Not every approach to formal verification can be fully automated, and thus, intervention and guidance by human experts may be necessary. Finally, *financial resources* are an issue in verification. Of course, this aspect can be regarded just as the subsumption of computational and human resources. However, there are evidently more financial issues if we consider verification as an integral part of the software development process. The costs for repairing errors detected by verification increase with every further stage of system development *by orders of magnitude*. Thus, verification techniques that can already be applied in early phases of software design, can also significantly reduce financial efforts.

In this work we develop a verification framework for concurrent systems based on *model checking* [41, 8], a formal method technique that approaches the aforementioned efforts. As the name implies, model checking is based on the exploration of a formal *model* that characterises the possible behaviour of the considered system. The final software code of the system is not a prerequisite for model checking. Instead, models may be based on partial specifications or on system prototypes. Hence, model checking can be used to discover errors in early design stages of the software development process where corrections are relatively *cheap*. Moreover, model checking is a *fully automatic* verification technique. The exploration of the model requires no intervention by a user, and usually, even the model generation is automated. Thus, model checking can considerably reduce the verification effort with

regard to *financial* and *human resources*. – Coping with the *computational demands* remains as the main challenge in model checking. There exist a number of effective approaches for reducing the complexity of verification. These techniques are commonly based on *abstraction*. Abstraction in the context of verification means to construct a small (i.e. abstract) model of the considered system that preserves the validity (and invalidity) of certain properties of interest. Our approach to abstraction-based verification will be extensively discussed in Chapter 4 of this thesis.

Beforehand, we want to take a look at the fundamental characteristics of model checking. Given a mathematical state space model of a software system and a formal description of its correctness requirements, model checking automatically verifies whether the model satisfies the requirements, or whether a violation can be detected. In the following, we give an elementary introduction to the formal models and specification formalisms used in model checking, and thus, provide the necessary background for our verification framework. We start with the classical temporal logic model checking. Subsequently, we will consider the three-valued generalisation of classical model checking, which serves as the basis of our later introduced approach to three-valued abstraction.

2.1 Classical Temporal Logic Model Checking

Temporal logic model checking is a fully automatic verification technique. It refers to the question of whether a model of a software system satisfies certain temporal logic correctness requirements. In common approaches, a *Kripke structure* [90] is used to model the state space of the system under consideration.

Definition 2.1 (Kripke Structure).
A *Kripke structure* over a set of atomic predicates AP is a tuple $K = (S, R, L, \mathbb{F})$ where

- S is a finite set of states,
- $R : S \times S \to \{true, false\}$ is a total transition function, i.e. $\forall s \in S : \exists s' \in S : R(s,s') = true$,
- $L : S \times AP \to \{true, false\}$ is a labelling function that associates a truth value with each predicate in each state,
- $\mathbb{F} \subseteq \mathbb{P}(R^{-1}(\{true\}))$ is a set of fairness constraints where each constraint $F \in \mathbb{F}$ is a set of *true* transitions.

A path π of a Kripke structure K is an infinite sequence of states $s_0 s_1 s_2 \ldots$ with $R(s_i, s_{i+1}) = true$. π_i denotes the i-th state of π and Π_s denotes the set of all

paths starting in $s \in S$. A path π is *fair* if it takes infinitely often a transition from every $F \in \mathbb{F}$. By Π_s^{fair} we denote the set of all fair paths starting in $s \in S$. Fairness constraints in Kripke structures are used to rule out unrealistic (i.e. unfair) behaviour of the modelled system. In Chapter 3 we will provide more details on unrealistic system behavior, and moreover, show how fairness constraints can be derived from a given system. Henceforth, we solely focus on fair paths. For illustration of the aforementioned definitions, we consider the Kripke structure K_1 in Figure 2.1.

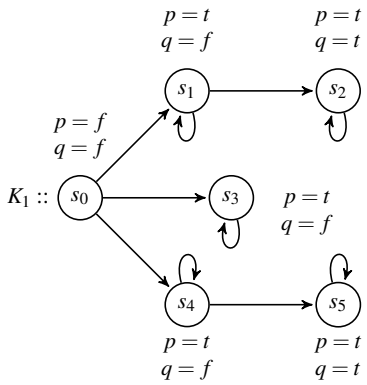

Fig. 2.1 Kripke structure K_1 over $AP = \{p,q\}$ with fairness constraint $\mathbb{F} = \{F\}$ where $F = \{(s_2,s_2),(s_3,s_3),(s_5,s_5)\}$. In the labelling, t abbreviates *true*, and f abbreviates *false*.

As we can see, the infinite sequence $\pi = s_0s_1s_1s_1\ldots$ is a path of K_1. However, since π does not take any transition from the set F infinitely often, this path is not fair. A fair path of K_1 is e.g. $\pi' = s_0s_1s_2s_2\ldots$.

Requirements, i.e. desirable properties of systems represented as Kripke structures can be formalised in temporal logic, an extension of the classical propositional logic. The *computation tree logic* (CTL) [37] is a branching-time logic for specifying such properties. The syntax of CTL can be defined in two steps. We can distinguish CTL *state* and *path* formulae:

Definition 2.2 (Syntax of CTL).
Let AP be a set of atomic predicates. The syntax of CTL *state formulae* is given by the following grammar:

$$\psi ::= p \mid \neg\psi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \mathbf{E}\phi \mid \mathbf{A}\phi$$

where $p \in AP$ and ϕ is a CTL path formula. Thus, state formulae permit the logical connectives $\neg, \wedge, \vee$ as well as the existential (**E**) or universal (**A**) quantification over path formulae. The syntax of CTL *path formulae* is given by the following grammar:

$$\phi ::= \mathbf{X}\psi \mid \mathbf{F}\psi \mid \mathbf{G}\psi \mid \psi\mathbf{U}\psi$$

where ψ is a CTL state formula. State formulae refer to properties of states and their branching structure, whereas path formulae characterise temporal properties of paths. As temporal operators, we have *next* (**X**), *eventually* (**F**), *globally* (**G**) and *until* (**U**). The formal semantics of these operators follows from Definition 2.3. In the evaluation of CTL formulae on Kripke structures only state formulae are considered – which, however, may be composed of path formulae. Henceforth, we refer to CTL state formulae just as *CTL formulae*.

Definition 2.3 (Fair Evaluation of CTL).
Let $K = (S, R, L, \mathbb{F})$ be a Kripke structure over a set of atomic predicates AP. Then the *fair evaluation* of a CTL formula ψ in a state s of K, written $[K, s \models \psi]$, is inductively defined as follows

$$\begin{aligned}
[K,s \models p] &:= \bigvee_{\pi \in \Pi_s^{fair}} L(\pi_0, p) \\
[K,s \models \neg\psi] &:= \bigvee_{\pi \in \Pi_s^{fair}} \neg\,[K, \pi_0 \models \psi] \\
[K,s \models \psi \wedge \psi'] &:= \bigvee_{\pi \in \Pi_s^{fair}} [K,\pi_0 \models \psi] \ \wedge\ [K,\pi_0 \models \psi'] \\
[K,s \models \psi \vee \psi'] &:= \bigvee_{\pi \in \Pi_s^{fair}} [K,\pi_0 \models \psi] \ \vee\ [K,\pi_0 \models \psi'] \\
[K,s \models \mathbf{EX}\psi] &:= \bigvee_{\pi \in \Pi_s^{fair}} [K,\pi_1 \models \psi] \\
[K,s \models \mathbf{EG}\psi] &:= \bigvee_{\pi \in \Pi_s^{fair}} \bigwedge_{i \in \mathbb{N}} [K,\pi_i \models \psi] \\
[K,s \models \mathbf{E}(\psi\mathbf{U}\psi')] &:= \bigvee_{\pi \in \Pi_s^{fair}} \bigvee_{i \in \mathbb{N}} \left([K,\pi_i \models \psi'] \wedge \bigwedge_{0 \le j < i} [K,\pi_j \models \psi] \right)
\end{aligned}$$

If $[K, s \models \psi]$ evaluates to *true* then the system modelled by K satisfies the property formalised by ψ. In case the evaluation yields *false*, the modelled system violates the property. The evaluation of the remaining CTL operators can be derived by the following equivalences

$$\mathbf{EF}\psi \equiv \mathbf{E}(\mathit{true}\mathbf{U}\psi),\ \mathbf{AF}\psi \equiv \mathbf{A}(\mathit{true}\mathbf{U}\psi),$$

$$\mathbf{EG}\psi \equiv \neg\mathbf{AF}\neg\psi,\ \mathbf{AG}\psi \equiv \neg\mathbf{EF}\neg\psi,$$

$$\mathbf{AX}\psi \equiv \neg\mathbf{EX}\neg\psi$$

where two CTL formulae ψ_1, ψ_2 are equivalent, written $\psi_1 \equiv \psi_2$, iff for all Kripke structures K and for all states s of K: $[K, s \models \psi_1] = [K, s \models \psi_2]$.

For our example Kripke structure K_1 we e.g. have that $[K_1, s_0 \models \mathbf{AF}p]$ yields *true*, i.e. for all fair paths starting in s_0 eventually p holds. It is sufficient to consider the substructure π_{K_1} of K_1 depicted in Figure 2.2 in order to validate $[K_1, s_0 \models \mathbf{AF}p]$. Such a substructure that proves the validity of a temporal logic formula ψ is called a *witness* for ψ.

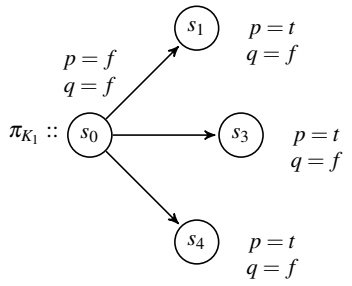

Fig. 2.2 Witness π_{K_1} for $[K_1, s_0 \models \mathbf{AF}p]$ in the Kripke structure K_1.

As another example, $[K_1, s_0 \models \mathbf{AG}(\mathbf{AF}(\mathbf{EX}q))]$ evaluates to *false*. This can be disproved by the fair path $\pi'_{K_1} = s_0 s_3 s_3 \ldots$ in K_1, a path where at some time p never holds next. We call such a path that refutes a CTL formula ψ a *counterexample* for ψ. Counterexamples for temporal logic properties are not necessarily paths, i.e. *linear* traces. Since CTL is a branching-time logic, counterexamples may also have a tree-like structure. Moreover, there is a duality between witnesses and counterexamples. A substructure π of a Kripke structure is a *counterexample* for a temporal logic formula ψ if and only if π is a *witness* for the formula $\neg\psi$. Thus, the tree-like witness π_{K_1} for $\mathbf{AF}p$ is also a counterexample for the negated property $\mathbf{EG}\neg p$, and the linear counterexample π'_{K_1} for $\mathbf{AG}(\mathbf{AF}(\mathbf{EX}q))$ is a witness for $\mathbf{EF}(\mathbf{EG}(\mathbf{AX}\neg q))$.

Conversely to the equivalence between CTL formulae, we have that two states of Kripke structures are equivalent with respect to the branching-time logic iff they satisfy the same set of CTL formulae. Such an equivalence relation on states of Kripke structures is denoted as a bisimulation.

Definition 2.4 (Bisimulation).
Let $K_1 = (S_1, R_1, L_1, \mathbb{F}_1)$ and $K_2 = (S_2, R_2, L_2, \mathbb{F}_2)$ be two Kripke structures, both defined over the same set of atomic predicates AP. Then a *bisimulation* between K_1 and K_2 is the greatest relation $\sim_b \subseteq S_1 \times S_2$ such that $s_1 \sim_b s_2$ implies

- $\forall p \in AP : L_1(s_1, p) = L_2(s_2, p)$,
- $\forall s'_1 \in S_1$ such that $R_1(s_1, s'_1)$ there is a state $s'_2 \in S_2$ with $R_2(s_2, s'_2)$ and $s'_1 \sim_b s'_2$,
- $\forall s'_2 \in S_2$ such that $R_2(s_2, s'_2)$ there is a state $s'_1 \in S_1$ with $R_1(s_1, s'_1)$ and $s'_1 \sim_b s'_2$.

We say, the Kripke structures K_1 and K_2 are *bisimilar*, denoted by $K_1 \sim_b K_2$, if there exists such a bisimulation between them. Moreover, two paths π^1 in K_1 and π^2 in K_2 are bisimilar iff $\forall k \in \mathbb{N} : \pi^1_k \sim_b \pi^2_k$. Then we also say, the path π^1 simulates the path π^2 and vice versa. Since bisimulation is reflexive,

transitive and symmetric, it is an equivalence relation. So far, there is no notion of fairness in this basic definition of a bisimulation relation. However, a bisimulation can be easily extended with fairness constraints [41]:

Definition 2.5 (Fair Bisimulation).
Let $K_1 = (S_1, R_1, L_1, \mathbb{F}_1)$ and $K_2 = (S_2, R_2, L_2, \mathbb{F}_2)$ be two Kripke structures, both defined over the same set of atomic predicates AP. Then a *fair bisimulation* between K_1 and K_2 is the greatest relation $\sim_b \subseteq S_1 \times S_2$ such that $s_1 \sim_b s_2$ implies

- $\forall p \in AP : L_1(s_1, p) = L_2(s_2, p)$.
- For every fair path $\pi^1 \in \Pi_{s_1}^{fair}$ in K_1 exists a fair path $\pi^2 \in \Pi_{s_2}^{fair}$ in K_2 such that $\forall k \in \mathbb{N} : \pi_k^1 \sim_b \pi_k^2$.
- For every fair path $\pi^2 \in \Pi_{s_2}^{fair}$ in K_2 exists a fair path $\pi^1 \in \Pi_{s_1}^{fair}$ in K_1 such that $\forall k \in \mathbb{N} : \pi_k^1 \sim_b \pi_k^2$.

Two Kripke structures are *fair bisimilar* if there exists such a fair bisimulation between them. Henceforth, we just write *bisimulation* when we refer to the fair bisimulation. In Figure 2.3 we see a Kripke structure K_2 that is bisimilar to the structure K_1 in Figure 2.1.

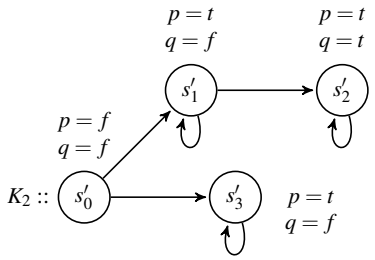

Fig. 2.3 Kripke structure K_2 over $AP = \{p, q\}$ with fairness constraint $\mathbb{F} = \{F\}$ where $F = \{(s'_2, s'_2), (s'_3, s'_3)\}$.

The bisimulation between K_1 and K_2 is defined by $\sim_b = \{(s_0, s'_0), (s_1, s'_1), (s_2, s'_2), (s_3, s'_3), (s_4, s'_1), (s_5, s'_2)\}$. We have that the path $\pi^2 = s'_0 s'_1 s'_2 \ldots$ in K_2 simulates the paths $\pi^1 = s_0 s_1 s_2 \ldots$ and $\pi'^1 = s_0 s_4 s_5 \ldots$ in K_1 and vice versa. Moreover, we can observe that K_2 is significantly smaller, with respect to the number of states and transitions, than the bisimilar K_1. From [41] we get the following result for bisimilar Kripke structures:

Theorem 2.1.
Let $K_1 = (S_1, R_1, L_1, \mathbb{F}_1)$ and $K_2 = (S_2, R_2, L_2, \mathbb{F}_2)$ be two bisimilar Kripke structures and let $\sim_b \subseteq S_1 \times S_2$ be the respective bisimulation. Moreover, let $s_1 \in S_1$,

$s_2 \in S_2$. *Then*

$$s_1 \sim_b s_2 \textit{ iff } (\forall \textit{ CTL formulae } \psi : [K_1, s_1 \models \psi] = [K_2, s_2 \models \psi]).$$

This theorem can be exploited to reduce the complexity of temporal logic model checking. Assume there is a verification task given by $[K_1, s_1 \models \psi]$. Then the general approach is to find a smaller Kripke structure K_2 with $K_1 \sim_b K_2$, a state s_2 in K_2 with $s_1 \sim_b s_2$, and then evaluate $[K_2, s_2 \models \psi]$. By Theorem 2.1 the obtained result can be transferred to the original verification task. Due to the smaller K_2 this approach is usually more efficient than directly evaluating $[K_1, s_1 \models \psi]$. For our running example with the Kripke structure K_2 in Figure 2.3 we have that $[K_2, s_0' \models \mathbf{AF}p] = \mathit{true}$ and $[K_2, s_0' \models \mathbf{AG}(\mathbf{AF}(\mathbf{EX}q))] = \mathit{false}$, which is compliant with the results obtained for the larger bisimilar Kripke structure K_1 in Figure 2.1.

However, given a Kripke structure K_1, then the smallest bisimilar K_2 might be still too large for an efficient verification. A less restrictive relation on Kripke structures is the simulation.

Definition 2.6 (Simulation).
Let $K_1 = (S_1, R_1, L_1, \mathbb{F}_1)$ and $K_2 = (S_2, R_2, L_2, \mathbb{F}_2)$ be two Kripke structures, both defined over the same set of atomic predicates AP. Then a *simulation* between K_1 and K_2 is the greatest relation $\preceq_s \subseteq S_1 \times S_2$ such that $s_1 \preceq_s s_2$ implies

- $\forall p \in AP : L_1(s_1, p) = L_2(s_2, p)$,
- $\forall s_1' \in S_1$ such that $R_1(s_1, s_1')$ there is a state $s_2' \in S_2$ with $R_2(s_2, s_2')$ and $s_1' \preceq_s s_2'$,

As we can see, a bisimulation corresponds to a simulation which additionally relates transitions of K_2 to transitions of K_1. Hence, every bisimulation is also a simulation. The basic definition of simulation can be extended with fairness constraints [41]:

Definition 2.7 (Fair Simulation).
Let $K_1 = (S_1, R_1, L_1, \mathbb{F}_1)$ and $K_2 = (S_2, R_2, L_2, \mathbb{F}_2)$ be two Kripke structures, both defined over the same set of atomic predicates AP. Then a *fair simulation* between K_1 and K_2 is the greatest relation $\preceq_s \subseteq S_1 \times S_2$ such that $s_1 \preceq_s s_2$ implies

- $\forall p \in AP : L_1(s_1, p) = L_2(s_2, p)$.
- For every fair path $\pi^1 \in \Pi_{s_1}^{fair}$ in K_1 there exists a fair path $\pi^2 \in \Pi_{s_2}^{fair}$ in K_2 such that $\forall k \in \mathbb{N} : \pi_k^1 \preceq_s \pi_k^2$.

Henceforth, we just write *simulation* when we refer to the fair simulation. If there exists such a relation $\preceq_s$ between K_1 and K_2 then we say, K_1 is simulated by K_2 or, conversely, K_2 simulates K_1. Simulation is not an equivalence relation

but a preorder. Thus, it is reflexive, transitive but not symmetric. For two states s_1 in K_1 and s_2 in K_2 with $s_1 \preceq_s s_2$ every path starting in s_1 can be simulated by a path starting in s_2, but *not* vice versa. Hence, CTL properties are generally not preserved under simulation. Nevertheless, we will see that simulation preserves properties from the *universal fragment of CTL* (ACTL). ACTL is restricted to universal quantification, and moreover, negation is solely permitted for atomic predicates.

Definition 2.8 (Syntax of ACTL).
Let AP be a set of atomic predicates. The syntax of ACTL *state formulae* is given by the following grammar:

$$\psi ::= p \mid \neg p \mid \psi \wedge \psi \mid \psi \vee \psi \mid \mathbf{A}\phi$$

where $p \in AP$ and ϕ is a CTL path formula.

The semantics of ACTL is the same as for CTL. Note that we have the same equivalences as before, and thus, an ACTL formula can be transferred into an equivalent CTL formula which may contain existential quantification as well. From [41] we get the following result:

Theorem 2.2.
Let $K_1 = (S_1, R_1, L_1, \mathbb{F}_1)$ and $K_2 = (S_2, R_2, L_2, \mathbb{F}_2)$ be two Kripke structures with $K_1 \preceq_s K_2$ and let $\preceq_s \subseteq S_1 \times S_2$ be the respective simulation. Moreover, let $s_1 \in S_1$, $s_2 \in S_2$. Then

$$s_1 \preceq_s s_2 \text{ iff } (\forall \text{ ACTL formulae } \psi : [K_2, s_2 \models \psi] \Rightarrow [K_1, s_1 \models \psi]).$$

Thus, given a verification task $[K_1, s_1 \models \psi]$, a common approach is to find a smaller Kripke structure K_2 that simulates K_1, a state s_2 in K_2 with $s_1 \preceq_s s_2$, and then evaluate $[K_2, s_2 \models \psi]$. In case $[K_2, s_2 \models \psi]$ yields *true*, this result can be transferred to the original verification task, whereas a *false* result for K_2 tells us nothing about K_1. However, since the simulation relation is less restrictive than the bisimulation, finding a small Kripke structure that simulates the original one is usually easier than finding a bisimilar Kripke structure. And moreover, many verification tasks can already be successfully accomplished under simulation.

We want to consider an example for such a simulation. In Figure 2.4 we have a Kripke stucture K_3 that simulates the structure K_2 from Figure 2.3 – and due to transitivity also K_1 from Figure 2.1. The simulation between K_2 and K_3 is defined by $\preceq_s = \{(s'_0, s''_0), (s'_1, s''_1), (s'_2, s''_2), (s'_3, s''_1)\}$. Again we get $[K_3, s''_0 \models \mathbf{AF}p] = true$, which conforms to the results obtained for the simulated Kripke structures K_1 and K_2. However, for the *non*-ACTL formula $\mathbf{AG}(\mathbf{AF}(\mathbf{EX}q))$ we have $[K_3, s''_0 \models \mathbf{AG}(\mathbf{AF}(\mathbf{EX}q))] = true$, which is not compliant with our former results for K_1 and K_2. This illustrates that under a simulation $K_2 \preceq_s K_3$ there might be feasible paths in K_3 that are *not* feasible in the simulated K_2. In our example, $\pi = s''_0 s''_1 s''_1 s''_1 \ldots$ in K_3 is such a *spurious* path.

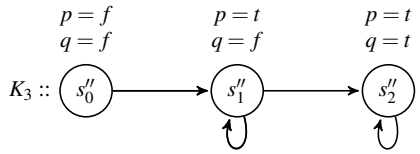

Fig. 2.4 Kripke structure K_3 over $AP = \{p,q\}$ with fairness constraint $\mathbb{F} = \{F\}$ where $F = \{(s''_1, s''_1), (s''_2, s''_2)\}$.

2.2 Three-Valued Temporal Logic Model Checking

Classical temporal logic model checking assumes that the considered system is *completely* known, and thus, can be properly modeled as a Kripke structure. This assumption, however, fails in several cases. System models might be incomplete at an early stage of design. Furthermore, the full (or even a similar) model of a complete system might be too large for verification so that only parts can be considered – which is in fact the case for the systems that are in the focus of this thesis. *Partially* known systems can be verified via *three-valued model checking* [22, 23]. This approach is based on an extended domain for truth values. In three-valued models, state properties and transitions can be *true*, *false* or *unknown*. For convenience, the additional truth value *unknown* is sometimes abbreviated by $\perp$. Partially known systems are modelled as three-valued Kripke structures, which generalise the classical Kripke structures.

Definition 2.9 (Three-Valued Kripke Structure).
A *three-valued Kripke structure* over a set of atomic predicates AP is a tuple $K = (S, R, L, \mathbb{F})$ where

- S is a finite set of states,
- $R : S \times S \to \{true, \perp, false\}$ is a total transition function, i.e. $\forall s \in S : \exists s' \in S : R(s,s') \in \{true, \perp\}$,
- $L : S \times AP \to \{true, \perp, false\}$ is a labelling function that associates a truth value with each predicate in each state,
- $\mathbb{F} \subseteq \mathbb{P}(R^{-1}(\{true, \perp\}))$ is a set of fairness constraints where each constraint $F \in \mathbb{F}$ is a set of *non-false* transitions.

Hence, a classical Kripke structure corresponds to a three-valued Kripke structure with $R^{-1}(\perp) = \emptyset$ and $L^{-1}(\perp) = \emptyset$. A path π of a three-valued Kripke structure K is an infinite sequence of states $s_0 s_1 s_2 \ldots$ with $R(s_i, s_{i+1}) \in \{true, \perp\}$. Thus, transitions now might take the additional truth value $\perp$. Apart from that, we use the same terminology as for paths of classical Kripke structures. An example for a three-valued Kripke structure is depicted in Figure 2.5.

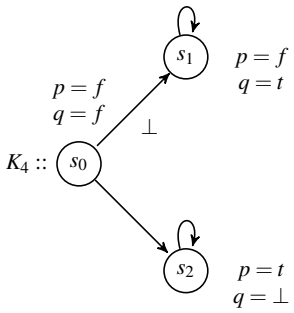

Fig. 2.5 Three-valued Kripke structure K_4 over $AP = \{p,q\}$ with fairness constraint $\mathbb{F} = \{F\}$ where $F = \{(s_1,s_1),(s_2,s_2)\}$.

As we can see, the transition (s_0,s_1), as well as the predicate q in the state s_2 have the indefinite truth value $\bot$. The sequence $\pi = s_0s_1s_1\ldots$ is a path of the three-valued Kripke structure K_4. Since π contains the *unknown* transition (s_0,s_1), we say π is an *unconfirmed path*. A *definite path* of K_4 is the trace $\pi' = s_0s_2s_2\ldots$ where all transitions along π' are definitely *true*.

Due to the extended domain for truth values, the evaluation of temporal logic formulae can no longer be based on classical propositional logic. Thus, three-valued model checking operates under the Kleene logic $\mathbb{K}_3$ [62] which generalises the propositional logic. The semantics of $\mathbb{K}_3$ is given by the truth tables in Figure 2.6.

$\wedge$	*true*	$\bot$	*false*
true	*true*	$\bot$	*false*
$\bot$	$\bot$	$\bot$	*false*
false	*false*	*false*	*false*

$\vee$	*true*	$\bot$	*false*
true	*true*	*true*	*true*
$\bot$	*true*	$\bot$	$\bot$
false	*true*	$\bot$	*false*

$\neg$	
true	*false*
$\bot$	$\bot$
false	*true*

Fig. 2.6 Truth tables for $\mathbb{K}_3$.

In three-valued model checking, CTL formulae have the same syntax as in the classical approach. The evaluation of formulae on Kripke structures now relies on the Kleene logic. The three-valued CTL semantics generalises the classical CTL semantics.

Definition 2.10 (Three-Valued Fair Evaluation of CTL).
Let $K = (S,R,L,\mathbb{F})$ be a three-valued Kripke structure over a set of atomic predicates AP. Then the *fair evaluation* of a CTL formula ψ in a state s of K, written $[K,s \models \psi]$, is inductively defined as follows

$$
\begin{aligned}
[K,s \models p] &:= \bigvee_{\pi \in \Pi_s^{fair}} L(\pi_0, p) \\
[K,s \models \neg\psi] &:= \bigvee_{\pi \in \Pi_s^{fair}} \neg [K,\pi_0 \models \psi] \\
[K,s \models \psi \wedge \psi'] &:= \bigvee_{\pi \in \Pi_s^{fair}} [K,\pi_0 \models \psi] \ \wedge \ [K,\pi_0 \models \psi'] \\
[K,s \models \psi \vee \psi'] &:= \bigvee_{\pi \in \Pi_s^{fair}} [K,\pi_0 \models \psi] \ \vee \ [K,\pi_0 \models \psi'] \\
[K,s \models \mathbf{EX}\psi] &:= \bigvee_{\pi \in \Pi_s^{fair}} R(\pi_0,\pi_1) \wedge [K,\pi_1 \models \psi] \\
[K,s \models \mathbf{EG}\psi] &:= \bigvee_{\pi \in \Pi_s^{fair}} \bigwedge_{i \in \mathbb{N}} (R(\pi_i,\pi_{i+1}) \wedge [K,\pi_i \models \psi]) \\
[K,s \models \mathbf{E}(\psi \mathbf{U} \psi')] &:= \bigvee_{\pi \in \Pi_s^{fair}} \\
&\quad \bigvee_{i \in \mathbb{N}} \left([K,\pi_i \models \psi'] \wedge \bigwedge_{0 \leq j < i} \left(R(\pi_j,\pi_{j+1}) \wedge [K,\pi_j \models \psi] \right) \right)
\end{aligned}
$$

As can be seen, the three-valued transition function R is included in the evaluation of some formulae. Furthermore, the three-valued labelling function L may introduce the additional truth value *unknown*. The evaluation of the remaining CTL operators can be again derived by the aforementioned equivalences. Checking a temporal logic formula on a three-valued Kripke structure now might yield *true*, *false* or *unknown*. For the Kripke structure K_4 in Figure 2.5 we e.g. have that $[K_4, s_0 \models \mathbf{EX}p]$ yields *true*, for $[K_4, s_0 \models \mathbf{AG}q]$ we obtain *false*, and $[K_4, s_0 \models \mathbf{AF}q]$ evaluates to *unknown*. For every *true* or *false* result obtained for a thee-valued Kripke structure K there exists at least one associated witness resp. counterexample. However, if model checking yields *unknown* then there is no substructure in K that definitely proves or refutes the temporal logic formula under consideration.

In the following we will see that an *unknown* result in verification reveals that the partially known system, or more specifically, the three-valued Kripke structure is too incomplete for a definite answer; but there exists at least one corresponding *unconfirmed counterexample* – a substructure with some *unknown* transitions or labellings that justifies the indefinite result. Moreover, we will see that if three-valued model checking returns *true* or *false*, then this result holds for the considered partially known system, or rather, for all possible completions, as well.

In a first step, we give a formal characterisation of the relation between *incomplete* and *more complete* systems, i.e. between *abstract* and *more concrete* Kripke structures. Therefore, we take a look at the Kleene logic $\mathbb{K}_3$ again. For $\mathbb{K}_3$ we have a *truth ordering* $\sqsubseteq_{\mathbb{K}_3}$ with $false \sqsubseteq_{\mathbb{K}_3} \bot \sqsubseteq_{\mathbb{K}_3} true$, and moreover an *information ordering* $\leq_{\mathbb{K}_3}$ (in words: 'less definite than') with $\bot \leq_{\mathbb{K}_3} true$, $\bot \leq_{\mathbb{K}_3} false$, and $true$, $false$ incomparable. Both orderings are reflexive. The information ordering can be used to define a preorder relation on three-valued Kripke structures [23]:

Definition 2.11 (Concreteness Preorder).
Let $K_a = (S_a, R_a, L_a, \mathbb{F}_a)$ and $K_c = (S_c, R_c, L_c, \mathbb{F}_c)$ be two three-valued Kripke structures, both defined over the same set of atomic predicates AP. Then a

concreteness preorder between K_a and K_c is the greatest preorder $\preceq_c \subseteq S_a \times S_c$ such that $s_a \preceq_c s_c$ implies

- $\forall p \in AP : L_a(s_a, p) \leq_{\mathbb{K}_3} L_c(s_c, p)$,
- $\forall s'_a \in S_a$ such that $R_a(s_a, s'_a) = true$ there is a state $s'_c \in S_c$ with $R_c(s_c, s'_c) = true$ and $s'_a \preceq_c s'_c$,
- $\forall s'_c \in S_c$ such that $R_c(s_c, s'_c) = \bot$ there is a state $s'_a \in S_a$ with $R_a(s_a, s'_a) = \bot$ and $s'_a \preceq_c s'_c$.

This basic definition of a concreteness preorder can be extended with fairness constraints [112]:

Definition 2.12 (Fair Concreteness Preorder).
Let $K_a = (S_a, R_a, L_a, \mathbb{F}_a)$ and $K_c = (S_c, R_c, L_c, \mathbb{F}_c)$ be two three-valued Kripke structures, both defined over the same set of atomic predicates AP. Then a *fair concreteness preorder* between K_a and K_c is the greatest preorder $\preceq_c \subseteq S_a \times S_c$ such that $s_a \preceq_c s_c$ implies

- $\forall p \in AP : L_a(s_a, p) \leq_{\mathbb{K}_3} L_c(s_c, p)$,
- For every fair path $\pi^a \in \Pi_{s_a}^{fair}$ in K_a exists a fair path $\pi^c \in \Pi_{s_c}^{fair}$ in K_c with $\forall k \in \mathbb{N}_{>0}$:

$$R_a(\pi^a_{k-1}, \pi^a_k) = true \;\; \Rightarrow \;\; R_c(\pi^c_{k-1}, \pi^c_k) = true \;\wedge\; \pi^a_k \preceq_c \pi^c_k$$

- For every fair path $\pi^c \in \Pi_{s_c}^{fair}$ in K_c exists a fair path $\pi^a \in \Pi_{s_a}^{fair}$ in K_a with $\forall k \in \mathbb{N}_{>0}$:

$$R_c(\pi^c_{k-1}, \pi^c_k) \neq false \;\; \Rightarrow \;\; R_a(\pi^a_{k-1}, \pi^a_k) \neq false \;\wedge\; \pi^a_k \preceq_c \pi^c_k$$

Henceforth, we just write *concreteness preorder* when we refer the fair concreteness preorder. If there exists such a concreteness preorder $\preceq_c$ between two Kripke structures K_a and K_c then we say, K_c is *more concrete* (or *less abstract*) than K_a, denoted by $K_a \preceq_c K_c$. We now consider again the three-valued Kripke structure K_4 in Figure 2.5. Remember that $[K_4, s_0 \models \mathbf{AF}q]$ evaluates to *unknown*. The path $\pi_{K_4} = s_0 s_2 s_2 \ldots$ justifies this result. As we can see, the predicate p is *unknown* in the state s_2. Hence, in a more concrete Kripke structure the predicate p is potentially *false* in s_2, and thus, π_{K_4} might be a real counterexample here. We call π_{K_4} an *unconfirmed counterexample*.

Definition 2.13 (Unconfirmed Counterexample).
Let $K = (S, R, L, \mathbb{F})$ be a three-valued Kripke structure over a set of atomic predicates AP. Moreover, let $s \in S$ be a state of K, and ψ be a CTL formula over AP with $[K, s \models \psi] = \bot$. Then an *unconfirmed counterexample* is a substructure π of K with some *unknown* transitions or labellings that justifies the indefinite

result; i.e. there exists a substitution of the $\bot$'s with definite truth values that extends π to a *real* counterexample for ψ.

Analogously, we can define *unconfirmed witnesses* as substructures that can be extended to real witnesses. However, for convenience we henceforth subsume both, unconfirmed counterexamples and witnesses under the term "unconfirmed counterexamples". Hence, in the remainder of this work we write *unconfirmed counterexample* when we refer to a substructure that can be extended to either a real counterexample or a real witness. (Unconfirmed) counterexamples provide explanations for the indefiniteness in three-valued model checking (or for requirement violations in classical model checking). In general, such counterexamples may not only comprise linear traces but also tree-like structures or even the entire Kripke structure. Thus, the generation an analysis of complete counterexamples is usually not feasible. Therefore, common approaches are either restricted to the verification very simple properties that can be refuted by inherently linear traces, or they return linear fragments of counterexamples that serve as partial explanations. We henceforth assume that an unconfirmed counterexample corresponds to a path, i.e. a structure that is linear by nature, or a linear fragment of a tree-like structure. Later we will see that in case of an *unknown* result in three-valued model checking, such unconfirmed counterexamples are particularly helpful for determining expedient concretisations of the underlying Kripke structure.

In Figure 2.7 we have two Kripke structures that are concretisations of the three-valued Kripke structure from Figure 2.5. As we can see, a con-

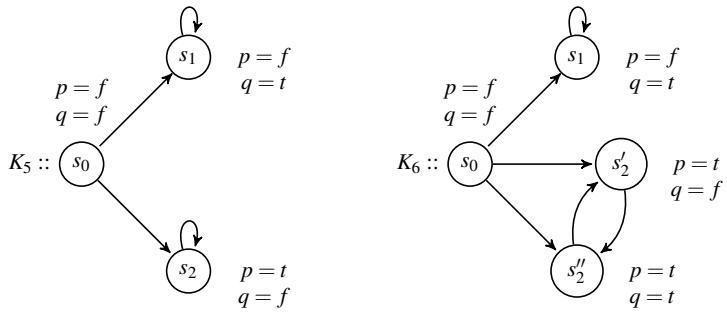

Fig. 2.7 Kripke structures K_5 and K_6 over $AP = \{p,q\}$ with fairness constraint $\mathbb{F} = \{F\}$ where $F = \{(s_1,s_1),(s_2,s_2)\}$ for K_5, and $F = \{(s_1,s_1),(s_2',s_2''),(s_2'',s_2')\}$ for K_6. Both, K_5 and K_6 are concretisations of the three-valued Kripke structure K_4 from Figure 2.5.

cretisation of a three-valued Kripke structure can be obtained by replacing *unknown* transitions and labellings by definite ones. We e.g. obtain K_5 from K_4 by substituting the *unknown* transition (s_0,s_1) with a corresponding *true* transition, and by labelling the state s_2 with $q = f$ instead of $q = \bot$. However, concretisation does not necessarily mean that the abstract and the concrete

model have to be structurally equal. The (larger) Kripke structure K_6 is also a feasible concretisation of the (smaller) K_4 in Figure 2.4. Thus, three-valued abstractions can also give us smaller representations of the modelled systems. We get the following theorem from [112]:

Theorem 2.3.
Let $K_a = (S_a, R_a, L_a, \mathbb{F}_a)$ and $K_c = (S_c, R_c, L_c, \mathbb{F}_c)$ be two three-valued Kripke structures with K_a more abstract than K_c and let $\preceq_c \subseteq S_a \times S_c$ be the respective concreteness preorder. Moreover, let $s_a \in S_a$, $s_c \in S_c$. Then

$$s_a \preceq_c s_c \text{ iff } \left(\forall \text{ CTL formulae } \psi : [K_a, s_a \models \psi] \leq_{\mathbb{K}_3} [K_c, s_c \models \psi]\right).$$

Hence, given two Kripke structures K_a and K_c with $K_a \preceq_c K_c$, then K_c represents more definite properties than K_a. However, any CTL formulae that evaluates to a definite value on K_a, evaluates to the same truth value on K_c. This result is exploited in several *abstraction-based* model checking techniques (e.g. [54, 67, 115]). Assume a verification task $[K_c, s_c \models \psi]$ where K_c is too large for directly applying model checking. Now, applying three-valued abstraction means to construct a more abstract (and thus, smaller) model K_a with $K_a \preceq_c K_c$ that is still concrete enough to obtain a definite result for $[K_a, s_a \models \psi]$ with $s_a \preceq_c s_c$. Unlike simulation-based model checking, the abstraction-based approach is not restricted to a fragment of CTL. Thus, both results $[K_4, s_0 \models \mathbf{EX}p] = true$ and $[K_4, s_0 \models \mathbf{AG}q] = false$ obtained for our running example, the abstract K_4, can be transferred to every possible concretisation, e.g. K_5 and K_6.

Abstraction in temporal logic model checking is a fundamental issue in this thesis, which will be addressed in Chapter 4. However, the capabilities of successfully applying abstraction highly depend on the characteristics of the considered system. Thus, in the next chapter we will give an introduction to *concurrent systems* – the kind of systems that are in the focus of the abstraction-based verification approach developed in this work. Furthermore, we will see how concurrent systems can be modelled as Kripke structures, and how typical correctness requirements of these systems can be specified in CTL. Beforehand, we provide a short overview of typical model checking techniques.

2.3 Model Checking Algorithms

So far, we have seen that Kripke structures can be used as state space representations of software systems. Moreover, correctness requirements of a system can be formalised as temporal logic formulae. – A model checker is a tool that takes a Kripke structure and a temporal logic formula as input, and then automatically verifies whether the formula holds for the structure or not. In this section, we want to take a closer look at the algorithmic aspects of model checking.

The basic algorithm for two-valued CTL model checking was originally introduced by Clarke and Emerson [36]. In the first step, this procedure constructs a parse tree representation of the input formulae ψ where each node of the tree represents a subformula of ψ. Then, starting at the level of atomic formulae (i.e. leaf nodes), for each subformula ψ' the set of states of the Kripke structure where ψ' holds is computed. The algorithm proceeds in a bottom-up manner until the root node, which represents the overall input formula ψ, is reached. CTL model checking under fairness additionally requires to compute the set of states where fair paths start. The algorithm has a time complexity of $O(|\psi| \times (|S| + |R|) \times |\mathbb{F}|)$. A detailed description of CTL model checking can e.g. be found in [41] and [8].

The previously outlined model checking algorithm relies on an *explicit* representation of the state space. Every single state of the underlying Kripke structure is explicitly enumerated during a model checking run. However, for larger state spaces such an explicit approach is hardly practicable. A usual way to overcome this issue is to work with a *symbolic* representation of the Kripke structure. Symbolic CTL model checking techniques are based on state space encodings that are substantially more compact than explicit Kripke structures. The most common approach to symbolic model checking relies on *binary decision diagrams* (BDDs) [25]. Binary decision diagrams are a data structures that are used to represent boolean functions. Since classical Kripke structures are based on a two-valued domain, they can be straightforwardly encoded as boolean functions. BDD representations of state spaces can be directly built from a system description. Thus, the expensive construction of an explicit Kripke structure is not required for applying symbolic model checking. The basic algorithm for BDD-based model checking was established by Burch et al. [26]. This algorithm proceeds in a similar manner as the explicit variant. However, the necessary computations can be performed much more efficiently based on binary decision diagrams than on explicit Kripke structures. Hence, BDD-based model checking allows for the verification of systems with far more than 10^{20} states. A prominent BDD-based model checker is part of the NuSMV framework for software verification, developed by Cimatti et al. [32]. More details on symbolic model checking with BDDs can be found in [41] and [8].

The two outlined algorithms for explicit and symbolic CTL model checking are tailored to a boolean setting. Nevertheless, model checking based on decision diagrams has also been applied in a multi-valued context. Multi-valued symbolic model checking, introduced by Chechik et al. [28], is a generalisation of the classical two-valued approach. It allows for the verification of models that are based on arbitrary multi-valued logics whose values form a finite quasi-boolean lattice. The algorithm for multi-valued model checking employs *multi-valued decision diagrams* (MDDs) [116] for state spaces encodings. These generalised decision diagrams also allow for compact representations of state spaces, and furthermore, for efficient model checking runs based on fast MDD operations. The time complexity of multi-valued model checking is

$O(|\mathbb{L}| \times |\psi| \times (|S| + |R|) \times |\mathbb{F}|)$ where $|\mathbb{L}|$ is the size of the underlying lattice. An existing symbolic model checker for multi-valued reasoning is χChek developed by Chechik et al. [30, 54]. The fundamentals of multi-valued symbolic model checking are extensively described in [28].

In this thesis, we pursue an approach to verification that is based on *three-valued* model checking – which is evidently a special case of multi-valued model checking. As we have already discussed in Section 2.2, thee-valued model checking is based on the Kleene logic $\mathbb{K}_3$ which forms the following quasi-boolean lattice:

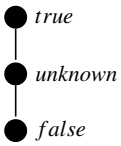

Fig. 2.8 Graphical representation of the finite quasi-boolean lattice $\mathbb{L}_{\mathbb{K}_3}$ corresponding to the Kleene logic $\mathbb{K}_3$.

This lattice has a size of 3 and thus in our approach model checking has a time complexity of $O(\mathbf{3} \times |\psi| \times (|S| + |R|) \times |\mathbb{F}|)$. The verification tool that we have implemented within this work is in fact build on top of the aforementioned multi-valued model checker χChek.

This nearly completes our background discussion. We have considered the logical and algorithmic aspects of model checking that are essential for *our* approach to verification. Finally, we want to provide a brief overview of *alternative* concepts from the field of model checking. CTL is not the only temporal logic that is employed for formalising correctness requirements in model checking. Another logic of relevance in verification is the *linear-time temporal logic* (LTL) [108]. LTL and CTL are incomparable with regard to their expressiveness. However, a large number of properties can be expressed in both logics. LTL model checking is based on different algorithms than CTL model checking. Explicit-state model checkers for LTL properties typically rely on automata-based computations [44]. A prominent example of such a model checker is SPIN [82]. Symbolic LTL model checking is commonly not based on BDD computations, but on boolean satisfiability solving: *Bounded model checking* (BMC) [21] is a variant of classical model checking that explores finite path prefixes rather than infinite paths. BMC can be reduced to *boolean satisfiability* (SAT) [49], and thus, verification can be efficiently performed by SAT-solvers. SAT-based bounded model checking for LTL properties is also supported by the NuSMV framework [33]. Model checking via satisfiability solving has also been considered in a three-valued context: In [126] the three-valued bounded model checking problem is reduced to two boolean satisfiability problems. The reduction of a multi-valued model checking problem to a number of classical two-valued model checking problems, e.g. [71, 24],

is a common alternative to direct approaches to multi-valued model checking like χChek [30, 54].

Chapter 3
Concurrent Systems

The verification framework that we have developed within this thesis focuses on *concurrent systems*, which are composed of many software processes running concurrently and communicating with each other. Such systems are in widespread practical use. A variety of examples can be found in the fields of network protocols and distributed computing. Due to the versatile concepts of *concurrency* and *communication*, concurrent systems are an appropriate and efficient choice for many complex computational tasks. However, it is exactly these concepts that make verification, i.e. proving the correctness of such systems, particularly challenging.

In this chapter, we give an introduction to the broad field of concurrent systems, which includes a definition of the syntax and semantics of such systems as well as a detailed description of the different concepts of communication. Moreover, we show how concurrent systems can be formally represented as control flow graphs and finally be transferred into a state space model for verification. We conclude this chapter with an overview of typical correctness requirements for concurrent systems, and we show how these requirements can be formalised in temporal logic.

3.1 Syntax and Semantics of Concurrent Systems

We start with a formal characterisation of concurrent software systems. A concurrent system Sys consists of $n \in \mathbb{N}$ processes $Proc_1$ to $Proc_n$ composed in parallel: $Sys = \|_{i=1}^{n} Proc_i$. It is defined over a set of system variables $Var = Var_s \cup \bigcup_{i=1}^{n} Var_i$ where Var_s is a set of shared variables and $Var_1, \ldots, Var_n$ are sets of local variables associated with the processes $Proc_1, \ldots, Proc_n$, respectively. Variables either have a basic type (e.g. *boolean*, *integer*) or an array type ($integer \rightarrow boolean$, $integer \rightarrow integer$). A process $Proc_i$ corresponds to a finite sequence of control locations $1_i, \ldots, k_i$ where each location is associated with an *operation* on the variable set $Var_s \cup Var_i$.

Definition 3.1 (Basic Operations).
Let $\{x_1,\ldots,x_m\}$ be a set of variables. The set of basic operations BOp on these variables consists of all statements of the form $assume(e) : x_1 := e_1, \ldots, x_m := e_m$ where $e, e_1, \ldots, e_m$ are expressions over $\{x_1,\ldots,x_m\}$.

Hence, every basic operation consists of an assume part, also called *guard*, and a list of assignments. Executing the guard $assume(e)$ blocks the execution of the assignment part until the boolean expression e evaluates to $true$. For convenience, we sometimes just write e instead of $assume(e)$. Moreover, we omit the assume part completely if e is constantly $true$.

A basic operation at some control location l is always followed by a reference to the subsequent location l'. Unless otherwise specified, we assume that $l' = l+1$. The current control location of a process $Proc_i$ can be regarded as the value of an additional local variable pc_i over the process' locations $Loc_i = \{1_i,\ldots,k_i\}$. Note that this *program counter* pc_i is *not* contained in the set of system variables Var. If we refer to *all* variables, then we explicitly write $Var \cup \bigcup_{i=1}^{n} \{pc_i\}$.

Beside basic operations, control locations may also be associated with *compound operations*. A compound operation consists of one or more sub-operations nested inside a control structure. An example is the *if-then-else* operation

$$l : \big(\textbf{if } e \textbf{ then } (l' : op_1) \textbf{ else } (l'' : op_2)\big)\ l''' :$$

where l, l', l'', l''' are control locations, e is a boolean expression and op_1 and op_2 are basic operations or itself compound operations. If e evaluates to $true$, then op_1 at location l' is selected for execution, else (if e evaluates to $false$) op_2 at l'' is selected. After the *if-then-else* operation has been executed, the process continues at location l'''. Further compound operations in our systems are, amongst others, *await*, *goto*, *for-to-do*, *while-do*, *do-while* and *loop-forever-do* with the usual semantics (which can be found e.g. in [94]). We denote the set of *all* (basic and compound) operations on the variables of a system by Op.

Initially, all processes of a concurrent system $Sys = \|_{i=1}^{n} Proc_i$ are at their first control location 1_i. Moreover, we assume that a deterministic initialisation of the system variables is given by an assertion φ_{Init} over Var. Now, a *computation* of a system (i.e. its behaviour over time) corresponds to a sequence where in each step one process is non-deterministically selected and the operation op at its current control location is attempted to be executed. In case the execution is not blocked by a guard, the system variables are updated according to the assignment part of op and the process advances to the consequent control location. The exact definition of a computation can be found in Section 3.3 where we introduce a formal model for concurrent systems. In our systems, computations are always *infinite*, and thus, every operation refers to a successor location. However, we are also able to simulate *termination* of a process by the terminal operation *end* which has the following semantics:

$$l: \textbf{ goto } l:$$

i.e. *end* at some control location l corresponds to an unconditional self-loop without an assignment. Hence, once a process has reached *end* in a computation, then every further selection of this (now terminated) process just triggers the self-loop.

In practice, the computation of a concurrent system is controlled by a scheduler which selects the next process, or rather operation, for execution. Schedulers commonly operate under *fairness*, i.e. they ensure that each process will eventually proceed. In this work, we therefore consider only fair, and thus, realistic computations. In our notion, a computation of a concurrent system is *fair* if and only if each process is infinitely often selected for executing an operation. Fairness plays a vital role in the verification of certain correctness requirements of concurrent systems. We discuss this aspect separately in Section 3.4.

To illustrate the definitions and terms with regard to concurrent systems, we consider the system Sys_1 in Figure 3.1 (written in a language similar to SPL [94]).

$$x: \textbf{integer where } x = 1$$

$$Proc_1 :: \begin{bmatrix} 1: & \textbf{loop forever do} \\ \big[2: & x := -x\big] \end{bmatrix} \parallel Proc_2 :: \begin{bmatrix} 1: & \textbf{while } x > 0 \textbf{ do} \\ \big[2: & \textbf{skip}\big] \\ 3: & \textbf{end} \end{bmatrix}$$

Fig. 3.1 Concurrent system $Sys_1 = Proc_1 \parallel Proc_2$ over $Var = Var_s = \{x\}$.

Here we have two processes composed in parallel and operating on the shared integer variable x. Process $Proc_1$ consists of a negation operation on x, nested inside an infinite loop. $Proc_2$ has a conditional *while*-loop (loop condition: x greater than 0); the *skip* inside the loop body denotes the *empty operation*, i.e. an operation with no guard and no assignment; the *while-loop* is followed by the terminal operation *end*.

A possible computation of Sys_1 is the sequence that steadily runs through the infinite loop of $Proc_1$, without being interrupted by an operation of $Proc_2$. However, this computation does not fulfil our notion of fairness. In a fair computation the execution of $Proc_1$ has to be infinitely often interrupted by $Proc_2$. Now, let us consider the following sequence of operations:

- $Proc_1$ executes the loop body *twice* in succession, i.e. it sets the value of x to -1 and then to 1 again
- $Proc_2$ evaluates the loop condition – which is currently *true* – and executes the loop body

The infinite repetition of these two steps corresponds to a fair computation of Sys_1 in which $Proc_2$ will never terminate. However, termination is generally

possible here, e.g. when $Proc_1$ executes the loop body only *once* before being interrupted by $Proc_2$. We see that the behaviour of concurrent systems (e.g. with regard to termination) crucially depends on the order of selected processes in a computation. Verifying a system, e.g. proving correct termination, is equivalent to considering *all* feasible computations. Our small example already gives us an idea about the potentially large number of different computations in concurrent systems, and thus, the complexity of its verification.

Another issue that is illustrated by Figure 3.1 is the fact that processes in a concurrent system may affect each other due to *shared variables*: The behaviour of $Proc_2$ depends on the value of the variable x which is modified by $Proc_1$. Because of concurrency $Proc_2$ may read x *before* or *after* the variable has been modified by $Proc_1$. Shared variables are the main reason for the large amount of possible computations in concurrent systems.

Now, we have a fundamental idea about the systems that are in the focus of our approach to verification. We have seen that shared variables enable a form of *communication* between processes. In the following, we introduce two advanced concepts of communication in concurrent systems. The first one is the variable type *semaphore* which can be used to control synchronisation between processes, i.e. to prevent undesired computations. And second, we introduce *communication channels* as an alternative to shared variables. Both of these concepts play a central role in many real-life concurrent systems.

3.1.1 Semaphores

In concurrent systems processes operate on shared resources. These resources can be simple variables but also complex data structures or hardware devices. For instance, consider a shared printer and a number of processes that attempt to perform a printing task. The printer has to be accessed exclusively, i.e. by only one process at a time. In general, this issue is known as the *mutual exclusion problem* which was first introduced by Dijkstra [50]:

> "A number of [...] processes [...] can be made in such a way that at any moment one and only one of them is engaged in the critical section of its cycle."

Here the *critical section* refers to a sequence of operations in which a process accesses an exclusive resource (e.g. the printer). In our systems we do not specify the access to exclusive resources explicitly. Instead, we use the operation *critical* to denote critical sections. According to the semantics of our systems, *critical* is just equivalent to the empty operation. However, it enables us to simply mark critical sections in processes without considering the internal details. Similarly, we mark unspecified non-critical sections by the (likewise empty) operation *non-critical*.

There are several approaches to solve the mutual exclusion problem in concurrent systems. One of the most prominent ones is based on *semaphores*.

A semaphore can be regarded as an integer variable with a limited set of operations on it. Commonly, for each exclusive resource one semaphore is introduced. The initial value of the semaphore, also called its *capacity*, denotes the number of processes that are allowed to access the associated resource at the same time. Now, for accessing the exclusive resource, i.e. for entering the corresponding critical section a process has to *acquire* a certain amount of the semaphores capacity. After finishing the operations on the exclusive resource the process can leave the critical section by *releasing* the acquired amount of capacity. For illustration, consider Sys_2 in Figure 3.2.

$$y : \textbf{semaphore where } y = 1$$

$$Proc_1 :: \left[\begin{array}{ll} 1: & \textbf{loop forever do} \\ & \left[\begin{array}{ll} 2: & \textbf{non-critical} \\ 3: & \textbf{acquire}(y,1) \\ 4: & \textbf{critical} \\ 5: & \textbf{release}(y,1) \end{array}\right] \end{array}\right] \parallel Proc_2 :: \left[\begin{array}{ll} 1: & \textbf{loop forever do} \\ & \left[\begin{array}{ll} 2: & \textbf{non-critical} \\ 3: & \textbf{acquire}(y,1) \\ 4: & \textbf{critical} \\ 5: & \textbf{release}(y,1) \end{array}\right] \end{array}\right]$$

Fig. 3.2 Concurrent system $Sys_2 = Proc_1 \parallel Proc_2$ over $Var = Var_s = \{y\}$.

Here we have two processes competing for access to a critical section – possibly associated with a printing task. The access is controlled by a shared semaphore variable. The semaphore operations *acquire* and *release* have the following semantics:

$$l : \textbf{acquire}(y,n) \quad \equiv \quad l : \langle \textbf{await } y \geq n;\ y := y - n \rangle$$

and respectively,

$$l : \textbf{release}(y,n) \quad \equiv \quad l : \ y := y + n$$

where y is a semaphore, n is a natural number, and the pointy brackets $\langle \ldots \rangle$ denote that the sequence of operations inside has to be executed in one atomic step. In our example, the capacity of the semaphore y is 1, and each process has to acquire the complete capacity before entering the critical section. Hence, at most one process can access the exclusive resource at the same time, which means mutual exclusion holds for the system Sys_2.

Semaphores are basic variables, and thus, they straightforwardly fit into our notion of concurrent systems with shared variables. Moreover, with semaphores we have a fundamental concept for establishing mutual exclusion, which is crucial in many verification problems.

3.1.2 Communication Channels

A more general form of synchronsation between processes can be achieved by introducing *communication channels*. The basic idea of communication chan-

nels is to shift the inter-process communication away from shared variables towards *message passing*. That is, processes solely communicate (i.e. synchronise) by sending and receiving messages via channels. Message passing has the advantage that it is independent from shared memory. Therefore, it is the preferred form of communication in distributed systems. However, we will see that communication channels can be simulated by a set of shared variables, and thus, message passing can be straightforwardly incorporated into our general notion of concurrent systems.

Definition 3.2 (Communication Channels).
A communication channel of length $n \in \mathbb{N}^+$ and type $t \in \{boolean, integer\}$ is given by a tuple $c = (buffer_c, rear_c, front_c, full_c, empty_c)$ where

- $buffer_c$: array $[n]$ of t
  an array representing the channels content,

- $rear_c$, $front_c$: integer
  pointer variables for the rear and front elements,

- $full_c$, $empty_c$: boolean
  boolean variables indicating whether the channel is full or empty.

We assume that communication channels in concurrent systems are initially empty, i.e. the initial configuration of a channel c is denoted by the expression

$$rear_c = 0 \ \wedge \ front_c = 0 \ \wedge \ \neg full_c \ \wedge \ empty_c.$$

A concurrent system is called a *message passing system* if and only if shared variables are solely channel-related and all other variables are local. Hence, in message passing systems inter-process communication can only be established by sending and receiving messages through channels. Our channels pass messages in first-in, first-out manner. Moreover, sending to full channels and receiving on empty channels will cause busy waiting. As an example, we consider the system in Figure 3.3.

$$c : \textbf{channel}\ [1]\ \textbf{of integer}$$

$$Proc_1 :: \begin{bmatrix} 1: & \textbf{send}(c,1) \\ 2: & \textbf{end} \end{bmatrix} \parallel Proc_2 :: \begin{bmatrix} \textbf{local}\ x : \textbf{integer where}\ x = 0 \\ 1: \quad \textbf{receive}(c,x) \\ 2: \quad \textbf{end} \end{bmatrix}$$

Fig. 3.3 Message passing system $Sys_3 = Proc_1 \parallel Proc_2$ over $Var = Var_s \cup Var_2 = \{c, x\}$.

Here we have two processes communicating via the channel c of type *integer* and length 1, i.e. c can buffer one integer value. Process $Proc_1$ attempts to send the value 1 to the channel c, whereas $Proc_2$ attempts to receive a value from c

and to store the received value in the local variable x. Since we have exactly one sender and one receiver in this system, both processes will eventually terminate. Note that for executing *receive*, $Proc_2$ has to wait until $Proc_1$ has send a value to the initially empty channel c. The exact semantics of the communication operations *send* and *receive* are given by:

$$\begin{aligned} l:\ \mathbf{send}(c,e) \ \equiv\ l:\ & \langle \mathbf{await}\ \neg full_c; \\ & buffer_c[rear_c] := e; \\ & empty_c := false; \\ & rear_c := rear_c + 1 \bmod n; \\ & full_c := (rear_c + 1 \bmod n = front_c)\rangle \end{aligned}$$

and respectively,

$$\begin{aligned} l:\ \mathbf{receive}(c,x) \ \equiv\ l:\ & \langle \mathbf{await}\ \neg empty_c; \\ & x := buffer_c[front_c]; \\ & full_c := false; \\ & front_c := front_c + 1 \bmod n; \\ & empty_c := (rear_c = front_c + 1 \bmod n)\rangle \end{aligned}$$

where c is a communication channel of length n, e is an expression of the same type as c and x is a variable, again of the same type as c. Hence, values transferred via channels have to be compatible with the channels type.

With message passing we have another form of inter-process communication that we want to consider in our approach to the verification of concurrent systems. Indeed, message passing is a concept of high practical relevance, especially in the field of distributed computing where no shared memory exists. Moreover, the introduction of communication channels involves several new issues with regard to the correctness of concurrent systems. For example, a processes might "starve" in front of a receive operation, because no communication partner is available. This and similar process synchronisation problems are in the focus of our verification technique.

3.2 Parameterised Systems

In the previous sections of this chapter we have introduced our notion of concurrent systems $Sys = \|_{i=1}^{n} Proc_i$. In our examples we solely considered systems with a fixed number n of processes. However, in practice, many systems are *paramterised* with regard to the number of processes: Network protocols for mutual exclusion, cache coherence or leader election are commonly defined for an *arbitrary* number of processes running in parallel. Verifying such

parameterised systems is particularly hard and even undecidable in the general case [7]. Nevertheless, these systems are in the focus of our verification technique, too. In this section we give a fundamental introduction to the field of parameterised systems. We start with a simple example:

$$y : \textbf{semaphore where } y = 1$$

$$\|_{i \in PID_N}\ Proc_i :: \left[\begin{array}{ll} 1: & \textbf{loop forever do} \\ & \left[\begin{array}{ll} 2: & \textbf{non-critical} \\ 3: & \textbf{acquire}(y,1) \\ 4: & \textbf{critical} \\ 5: & \textbf{release}(y,1) \end{array}\right] \end{array}\right]$$

Fig. 3.4 Parameterised system $Sys_4 = \|_{i \in PID_N}\ Proc_i$ over $Var = Var_s = \{y\}$ where PID_N is a set of process indices with a parameterised size $N \in \mathbb{N}$, e.g. $PID_N = \{1, \ldots N\}$.

The system Sys_4 in Figure 3.4 consists of N processes competing for access to a critical section. In the parametrised setting we assume that the capital N does not represent a fixed integer but a parameter, and thus, an unbounded number of processes might run in parallel. As we can see, Sys_4 is iteratively defined over the *process index i*, and each process $Proc_i$ executes the same sequence of operations. The processes only differ in their unique index value. Therefore we say that Sys_4 is *fully symmetric* with respect to process indices. Symmetry in paramterised systems is a characteristic that can be efficiently exploited for verification. Moreover, several real-life examples of parameterised systems are inherently symmetric, since their processes are commonly instances of one and the same process template. Hence, we also want to look at such *fully symmetric systems* in our approach to verification. In general, a fully symmetric system is defined as follows:

Definition 3.3 (Fully Symmetric System).
Let $Proc$ be a process defined over $Var_s \cup Var_l$ where Var_s is a set of shared variables and Var_l is a set of local variables. Then the corresponding *fully symmetric system* is defined as

$$Sys = \|_{i \in PID_N}\ Proc_i \text{ over } Var = Var_s \cup (Var_l \times PID_N)$$

where $N \in \mathbb{N}$ is a parameter of Sys and PID_N is a set of process indices of size N. Moreover, each $Proc_i$ is a replication of $Proc$ defined over $Var_i = Var_s \cup (Var_l \times i)$, i.e. $Proc_i$ is obtained from $Proc$ by preserving the control structure of $Proc$, and by replacing each basic operation bop in $Proc$ by $bop_i = bop[x/(x,i) \mid x \in Var_l]$.

Hence, all processes in Sys execute the same code and there exists a replication of the set of local variables Var_l for each process, i.e. the process indices are lifted to Var_l. Again, this fits into our general notion of concurrent systems, since we can rewrite $Var_l \times PID_N$ as $\bigcup_{i \in PID_N} Var_i$. Moreover, our understanding

of symmetry demands that each process has the same initial condition, i.e. that all replications of a variable from Var_l have the same initial value.

In the example system Sys_4 we have one shared semaphore variable y and no (explicit) local variables. However, each process in Sys_4 has a program counter ranging over identical locations, and thus, in a broader notion we can regard the program counter pc as a local variable with N replications: $pc \times PID_N$.

Full symmetry is a strong restriction on a parameterised system, because it demands that that *all* processes are identical. In fact, only few real-life systems fulfill this requirement. Nevertheless, in much more cases the processes of a parameterised system can at least be divided into *classes* of fully symmetric processes. We call such a system *class-wise symmetric*. A simple example is given below in Figure 3.5.

$$y : \textbf{semaphore where } y = N_{Rd}$$

$$\|_{i \in PID^{Rd}_{N_{Rd}}} Rd_i :: \left[\begin{array}{ll} 1: & \textbf{loop forever do} \\ \left[\begin{array}{ll} 2: & \textbf{non-critical} \\ 3: & \textbf{acquire}(y,1) \\ 4: & \textbf{critical} \\ 5: & \textbf{release}(y,1) \end{array}\right] & \end{array}\right] \quad \|_{j \in PID^{Wrt}_{N_{Wrt}}} Wrt_j :: \left[\begin{array}{ll} 1: & \textbf{loop forever do} \\ \left[\begin{array}{ll} 2: & \textbf{non-critical} \\ 3: & \textbf{acquire}(y,N_{Rd}) \\ 4: & \textbf{critical} \\ 5: & \textbf{release}(y,N_{Rd}) \end{array}\right] & \end{array}\right]$$

Fig. 3.5 Class-wise symmetric system $Sys_5 = \|_{i \in PID^{Rd}_{N_{Rd}}} Rd_i \,\|_{j \in PID^{Wrt}_{N_{Wrt}}} Wrt_j$ consisting of a reader class Rd and a writer class Wrt. $PID^{Rd}_{N_{Rd}}$ and $PID^{Wrt}_{N_{Wrt}}$ are sets of process indices with parameterised sizes $N_{Rd} \in \mathbb{N}$ resp. $N_{Wrt} \in \mathbb{N}$.

The system Sys_5 implements an algorithm for the *readers-writers problem* [45]. We have two classes of processes: *readers* Rd_i and *writers* Wrt_j. Multiple readers may enter the critical section at the same time, whereas if one writer is modifying data, no other process is allowed to access the critical section. The problem is solved via a semaphore with a capacity of N_{Rd}, which is actually the (parameterised) number of reader processes in the system. As we have two classes, the overall set of process indices PID_N is partitioned into $PID^{Rd}_{N_{Rd}}$ (readers) and $PID^{Wrt}_{N_{Wrt}}$ (writers). More generally, a class-wise symmetric system consisting of k classes is defined as follows:

Definition 3.4 (Class-Wise Symmetric System).
Let $\{Proc^1, \ldots, Proc^k\}$ be a set of processes where each $Proc^m$ $(1 \leq m \leq k)$ is defined over a set of variables $Var^m = Var^m_s \cup Var^m_l$ with $Var^1_l, \ldots, Var^k_l$ pairwise disjoint. We call the index m the *class* of a process $Proc^m$. Then, according to Definition 3.3, we obtain a corresponding fully symmetric system Sys^m for each class m:

$$Sys^m = \|_{i \in PID^m_{N_m}} Proc^m_i \text{ over } Var^m = Var^m_s \cup (Var^m_l \times PID^m_{N_m})$$

We assume that the sets $PID^1_{N_1}, \ldots, PID^k_{N_k}$ are pairwise disjoint, and thus, every process $Proc^m_i$ has a unique index. In addition, we assume that each class m has a dedicated program counter pc^m with a replication for each process in the class: $pc^m \times PID^m_{N_m}$. Now, the corresponding *class-wise symmetric system* is defined as

$$Sys = \|^k_{m=1} Sys^m \text{ over } Var = \bigcup_{m=1}^{k} \left(Var^m_s \cup (Var^m_l \times PID^m_{N_m})\right).$$

We explicitly allow that the variable sets $Var^1_s, \ldots, Var^k_s$ have common elements, i.e. communication between processes of different classes is permitted.

So far, we have seen that there exists a wide range of different kinds of concurrent systems in practical use. The systems can be distinguished by the underlying concept of communication, by symmetry characteristics, and whether they are finite or not. Nevertheless, we have introduced a general notion of concurrent systems, under which all these different kinds are comprised. In the next section we will show how our systems can be transferred into a computational model for verification.

3.3 Modelling Concurrent Systems

Verifying concurrent systems involves the exploration of the systems state space. Thus, we need to transfer our systems into a model that represents the set of reachable states under all computations. In Chapter 2 we have already introduced Kripke structures as the most common computational model in automatic verification. Here, we will see how states and the state space are formally defined for concurrent systems, and how a system can be transformed into a Kripke structure. In order to explore the state space of a concurrent system, we first of all require a formal characterisation of its control flow. In the previous section we have already described the control structure of single processes, but in an intuitive and rather informal way. Now, we show that concurrent systems can be straighforwardly transferred into *control flow graphs* [5].

Definition 3.5 (Processes as Control Flow Graphs).
Let $Proc_i$ be a process with operations from a set Op. Moreover, let BOp be the corresponding set of basic operations, i.e. for each basic operation $bop \in Op$: bop is also contained in Bop, and for each compound operation $op \in Op$: BOp contains all basic operations nested inside op. Then $Proc_i$ can be represented as a *control flow graph* (CFG) $G_i = (Loc_i, \delta_i)$, where Loc_i is the set of control locations of $Proc_i$ and $\delta_i \subseteq Loc_i \times BOp \times Loc_i$ is a labelled transition relation, i.e. transitions are labelled with basic operations.

Hence, basic operations of the form $l : bop\ l'$: can be directly mapped to the control flow graph: we get the corresponding transition $\delta_i(l, bop, l')$. For compound operations this mapping is a bit more complicated. Remember that every compound operation can be decomposed into a set of basic operations that are nested inside a control structure. This control structure corresponds to a subgraph of the CFG where the edges are labelled with the basic operations from the decomposition. As an example, we consider again the compound *if-then-else* operation:

$$1 :\ (\textbf{if}\ \ e\ \ \textbf{then}\ \ (2 :\ bop_1)\ \ \textbf{else}\ \ (3 :\ bop_2))\ 4 :$$

Assuming that bop_1 and bop_2 are basic operations, the corresponding control flow representation now looks as follows:

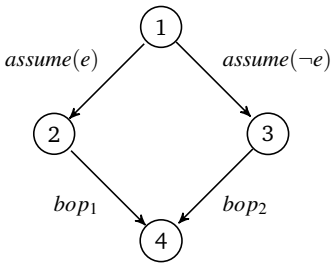

Fig. 3.6 Control flow representation of the *if-then-else* operation.

In a similar way we can transform all other compound operations into (subgraphs of) CFGs, which enables us to formally represent the control flow of individual processes. However, concurrent systems are composed of *many* processes running in parallel. Therefore, we need to define a notion of *parallel compositions* of control flow graphs.

Definition 3.6 (Concurrent Systems as Control Flow Graphs).
Let $Sys = \|_{i=1}^{n} Proc_i$ be a concurrent system, where each process $Proc_i$ is given as a control flow graph G_i. Then Sys can be represented as a composite control flow graph $G = (Loc, \delta)$. $Loc = \times_{i=1}^{n} Loc_i$ is the set of combined locations and $\delta \subseteq Loc \times BOp \times [1..n] \times Loc$ is a labelled transition relation with $\delta(l, bop, i, l') = \delta_i(l_i, bop, l_i')$, where l_i denotes the individual location of $Proc_i$ in the combined location $l = (l_1, \ldots, l_n)$.

Hence, each transition of a composite control flow graph is additionally labelled with the index $i \in [1..n]$ of the associated process. A fair computation of a concurrent system Sys corresponds to an infinite path in the composite control flow graph representing Sys, where for each $i \in [1..n]$ a transition $\delta(l, bop, i, l')$ occurs infinitely often.

For illustration, we consider the concurrent system Sys_6 in Figure 3.7 and show, how it can be transferred into a composite control flow graph. Sys_6

consists of two processes $Proc_1$ and $Proc_2$. Both attempt to execute one basic operation on a shared variable x and then terminate with the *end* operation.

$$x : \textbf{integer where } x = 1$$

$$Proc_1 :: \begin{bmatrix} 1: & x := x+1 \\ 2: & \textbf{end} \end{bmatrix} \parallel Proc_2 :: \begin{bmatrix} 1: & x := x+x \\ 2: & \textbf{end} \end{bmatrix}$$

Fig. 3.7 Concurrent system $Sys_6 = Proc_1 \parallel Proc_1$ over $Var = Var_s = \{x\}$.

In a first step, we transform the processes $Proc_1$ and $Proc_2$ of our concurrent system into single control flow graphs G_1 and G_2 (see Figure 3.8).

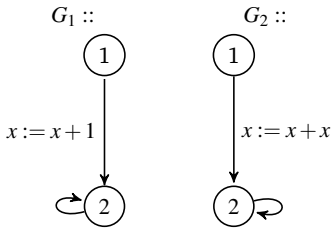

Fig. 3.8 Control flow graphs G_1 and G_2 of the processes $Proc_1$ and $Proc_2$.

In the second step, we build the composite control flow graph G for G_1 and G_2 (see Figure 3.9). We can see, that in each combined location there are two transitions enabled, i.e. each process can always be selected for executing its next operation. Every infinite path in G that eventually reaches location $(2,2)$ and then alternately takes the self-loops associated with $Proc_1$ and $Proc_2$, corresponds to a fair computation of Sys_6.

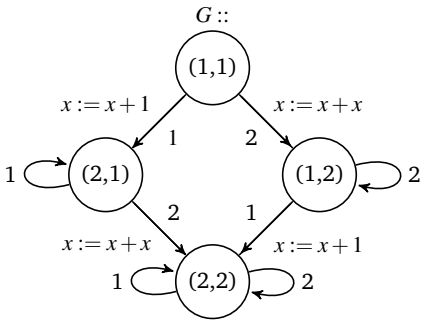

Fig. 3.9 Composite control flow graph G of the system $Sys_6 = Proc_1 \parallel Proc_2$.

Control flow graphs allow us to formally represent all sequences of operations that might occur during the execution of a concurrent system. For verifying a system, we moreover need to model the systems state space. A *state* of a concurrent system corresponds to a feasible valuation of all its variables, including the program counters. For instance, the initial state of the concurrent system in Figure 3.7 can be characterised by the tuple $s_{Init} = (pc_1 = 1, pc_2 = 1, x = 1)$. We denote the valuation of an expression e in a state s by $s(e)$, e.g. for s_{Init} we have $s_{Init}(x) = 1$, and $s_{Init}(x > 0) = true$. Now, the overall *state space* of a system corresponds to the set of all states over its variables. We write S_{Var} to denote the set of states over the system variables Var, and respectively, $S_{Var \cup \bigcup_{i=1}^{n} \{pc_i\}}$ to denote the set of states over the system variables *and* program counters.

In order to obtain a formal model of the state space, we transform our concurrent systems into Kripke structures (compare Chapter 2).

Definition 3.7 (Concurrent Systems as Kripke Structures).
Let $Sys = \|_{i=1}^{n} Proc_i$ be a concurrent system given by a composite control flow graph $G = (Loc, \delta)$. Moreover, let $Pred$ be a set of predicates (i.e. boolean expressions) over the system variables Var. The corresponding Kripke structure is a tuple $K = (S, R, L, \mathbb{F})$ over a set of atomic predicates $AP = Pred \cup \{pc_i = j \mid i \in [1..n], j \in Loc_i\}$ with

- $S := Loc \times S_{Var}$,
- $R(\langle l, s\rangle, \langle l', s'\rangle) := \bigvee_{i=1}^{n} R_i(\langle l, s\rangle, \langle l', s'\rangle) :=$
  $\bigvee_{i=1}^{n} (\delta(l, bop, i, l') \wedge s(e) \wedge s'(x_1) = s(e_1) \wedge \ldots \wedge s'(x_m) = s(e_m))$
  where $bop = assume(e) : x_1 := e_1, \ldots, x_m := e_m$,
- $L(\langle l, s\rangle, p) := s(p)$ for any $p \in Pred$,
- $L(\langle l, s\rangle, pc_i = j) := \begin{cases} true & \text{if } l_i = j \\ false & \text{else} \end{cases}$
  where l_i is the location of $Proc_i$ in the combined location l,
- $\mathbb{F} := \left\{ \{(s, s') \mid R_i(s, s') \neq false\}_{i \in [1..n]} \right\}$
  for each process $Proc_i$ a fairness set F_i with all associated transitions.

Hence, given a concurrent system Sys and a set of predicates $Pred$, we can construct the corresponding Kripke structure K. Now, a *fair computation* of Sys corresponds to a fair path π in K, i.e. an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ with $s_0 \models \varphi_{Init}$, for all $i \in \mathbb{N}$: $R(s_i, s_{i+1})$, and π satisfying the fairness requirements given by $\mathbb{F}$ (compare Chapter 2). As an example, the Kripke structure for the system Sys_6 in Figure 3.7. and the set of predicates $Pred = \{(x < 4)\}$ is shown in Figure 3.10.

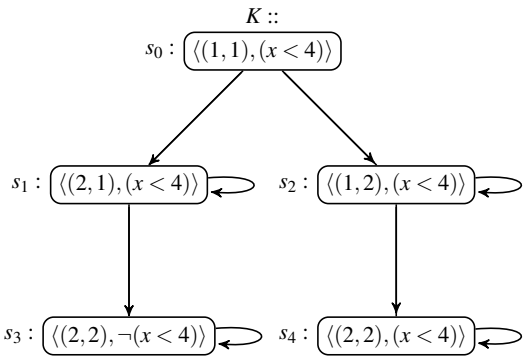

Fig. 3.10 Kripke structure K for the concurrent system Sys_6 and the set of predicates $Pred = \{(x < 4)\}$. The set of fairness constraints is $\mathbb{F} = \{F_1, F_2\}$ where $F_1 = \{(s_0,s_1),(s_1,s_1),(s_2,s_4),(s_3,s_3),(s_4,s_4)\}$ and $F_2 = \{(s_0,s_2),(s_1,s_3),(s_2,s_2),(s_3,s_3),(s_4,s_4)\}$. Only reachable states are depicted.

According to Definition 3.7, we can, in theory, transform any concurrent system into a computational model for temporal logic model checking. However, the number of states in a Kripke structure grows exponentially with the size of the modelled system. Thus, for real-life systems a straightforward modelling is practically not feasible. The additional application of abstraction techniques is typically necessary, which we will discuss in Chapter 4. – Furthermore, verifying a *parameterised* concurrent system means to check *all* (i.e. an infinite number of) instantiations of the system. Hence, parameterised verification is undecidable in general. Later we will show, that in many cases verification results obtained on single instantiations can be transferred to the overall parameterised system, i.e. to each possible instantiation.

Nevertheless, modelling the state space of parameterised – and in particular *symmetric* – systems deserves some additional remarks. Remember that in symmetric systems all processes execute the same code and there exists a replication of the set of local variables Var_l for each process. Hence, a local variable $x \in Var_l$ that is associated with some process $Proc_i$ can be represented as a tuple (x,i). Now, the valuation of (x,i) in a state s is denoted by $s(x,i)$. This lets us define local views on states of symmetric systems. For a state s of a symmetric system $Sys = \|_{i \in PID_N} Proc_i$ we write $s[i]$ to describe the *local view* of process $Proc_i$ on s, where $s[i](x) = s(x)$ for a shared variable $x \in Var_s$, and $s[i](x) = s(x,i)$ for a local variable $x \in Var_l$. These local views will later be very helpful when we exploit symmetry to verify parameterised systems (compare Chapter 6).

3.4 Correctness Requirements of Concurrent Systems

We conclude this chapter with a look at correctness requirements that we want to verify with our developed framework. Verifying a concurrent system is synonymous with showing its correctness in terms of certain specified requirements. A system is regarded as correct whenever it satisfies all of its requirements. Hence, proved correctness does not necessarily involve the absence of *any* kind of unfavourable behaviour. Verification rather facilitates the exclusion of specific serious errors that have been previously specified. In Chapter 2 we already introduced the temporal logic CTL as a logic for formalising requirements that can be verified via model checking. Subsequently, we will discuss typical correctness requirements of concurrent systems. Moreover, we will show how these requirements can be specified in CTL.

Correctness is indispensable for safety-critical systems. Failures of software used in aircrafts, power plants or medical systems may cause exceedingly high costs or may even threaten human lives. One major category of requirements for software systems is thus *safety*. A safety property can be informally characterised as "nothing bad will ever happen". Concurrent systems are, due to the large number of possible interleavings, particularly prone to safety errors. A *deadlock*, i.e. a situation where multiple processes wait for each other, is a typical example for the violation of a safety property. Another common safety property is *mutual exclusion*: Never more than one process shall enter a critical section at the same time. We have already seen in Section 3.1.1 that mutual exclusion can be established by means of a semaphore. The shared semaphore of the concurrent system Sys_2 in Figure 3.2 ensures that at any moment at most one process can enter the critical section. Now we want to see, how this property can be specified in temporal logic. The CTL formula corresponding to the mutual exclusion requirement for the two processes of Sys_2 is

$$\mathbf{AG}\neg((pc_1 = 4) \wedge (pc_2 = 4)).$$

This formula claims that *always globally* $Proc_1$ *and* $Proc_2$ are *not* simultaneously in their critical section, i.e. at location 4. A counterexample to this property thus would be a finite computation that starts in the initial state of the system and ends in a state where $(pc_1 = 4) \wedge (pc_2 = 4)$ holds. However, this safety property yields *true* for Sys_2 and thus there does not exist such a counterexample in Sys_2. Mutual exclusion is one of the most prominent examples for safety requirements of concurrent systems. Of course there exist a large number of other safety properties that can be expressed in CTL. Nevertheless, in this thesis we will mainly focus on mutual exclusion when we consider safety issues. The key characteristic that distinguishes safety properties from non-safety ones is that fact that a counterexample to a safety requirement can be always given by a *finite* computation, i.e. a finite path prefix. Hence, the possibility of refuting a property by a finite prefix can be regarded as a formal characterisation of safety.

Evidently, not every requirement that is expressible in CTL can be refuted by a finite computation. A notable category of such kind of properties is *liveness*. Informally, a liveness property is a requirement claiming that "something good will happen eventually". Liveness is particularly vital for concurrent systems with competing processes. Such a scenario may involve computations where some processes continuously proceed, whereas other processes are ignored all the time by the scheduler. Such a stagnation of certain processes in concurrent systems is a typical example for a violation of liveness. One can imagine that in safety-critical systems liveness errors can be as harmful as safety errors. For instance, a scheduler that permanently ignores a process for automotive braking control may cause great danger to the car driver. In Section 3.1 we already mentioned that for this reason concurrent systems are typically executed under reasonable fairness conditions that ensure progress for each process. However, fairness does not guarantee the absence of all liveness defects. For illustration, we again consider the concurrent system Sys_2 in Figure 3.2. A typical liveness requirement for concurrent systems with exclusive resources is that certain process, e.g. $Proc_1$, will always repeatedly access the critical resource, which can be formalised in CTL as follows:

$$\mathbf{AG}\,(\mathbf{AF}(pc_1 = 4))\,.$$

This property is not satisfied for Sys_2 under our notion of fairness. A fair computation demands that each process will always eventually proceed in terms of executing its next operation. However, the unsuccessful attempt to access the critical section by executing the operation $acquire(y,1)$ (when the semaphore is already acquired by another process) is also a form of progress. Hence, a fair *infinite* run where $Proc_2$ repeatedly accesses the critical section, whereas $Proc_1$ always attempts to acquire the semaphore "at the wrong moment", is thus a counterexample for $\mathbf{AG}(\mathbf{AF}(pc_1 = 4))$. It is obviously not possible to refute such a liveness property by a *finite* computation. In this thesis, we will mainly consider liveness requirements of the form $\mathbf{AG}(\mathbf{AF}(pc_1 = 4))$ (repetitive progress), or of the simpler form $\mathbf{AF}(pc_1 = 4)$ (eventual progress).

In literature there exist a number of different formal notions of liveness. Here we follow the notion of Alpern and Schneider [6] who characterise liveness requirements as properties that do not restrict finite behaviour, but require a condition on infinite behaviour. A survey of different characterisations of safety and liveness properties can be found in [89]. In fact, not every requirement that can be specified in temporal logic is either a safety or a liveness property. There exist properties that belong to neither of these categories. However, safety and liveness requirements are of prime importance in verification of safety-critical concurrent systems. Thus, in this work we will solely focus on these two kinds of properties. Moreover, we want to mention that there also exist different notions of fairness. Our fairness assumption is also known as *justice* or *weak fairness*. It can be formalised as follows: *Every basic operation (of a process) that is continuously enabled (i.e. its guard yields*

true) is eventually executed. Another notion of fairness is *compassion* alias *strong fairness*: *Every basic operation that is infinitely often enabled (but not necessarily continuously) is eventually executed.* Model checking under strong fairness involves a significantly higher complexity than model checking under weak fairness. A detailed survey of fairness notions can be found in [92].

So far, we have seen how concurrent systems can be transferred into a formal state space model – which generally enables verification. Moreover, we have discussed a number of correctness requirements that are of particular interest in the verification of concurrent systems. Such requirements can be formalised in temporal logic and automatically verified via model checking. However, the full state space of real-life systems often widely exceeds the capacity of todays verification tools. In the next chapter, we will see how the state space of a concurrent system can be reduced, and thus, verification can be made feasible, by the application of abstraction techniques.

Chapter 4
Abstraction for Concurrent Systems

The verification of software systems via model checking is generally performed in two stages. In a first step, the system is transferred into a state space model and the system requirements are specified in a formal language. Next, the obtained model is exhaustively explored in order to prove correctness or to find a violation of the requirements. In the previous chapter we introduced instructions for straightforwardly constructing the concrete state space model of a concurrent system. However, for most real-life systems the concrete state space is exorbitantly large, and thus, its construction would require an unfeasibly large amount of computational resources. Hence, a common approach in verification is to build an *abstract* model of the system without ever considering the corresponding concrete state space. Although abstraction inherently involves a loss of information about the original system, an abstract model can be still precise enough to successfully perform verification – *without* suffering from the state space complexity.

In this chapter, we present the abstraction part of our developed verification framework for concurrent systems. We start with the introduction of the two core concepts of our approach: *predicate abstraction*, a well-established technique for reducing the complexity of formal verification, and moreover, *spotlight abstraction*, an extension of predicate abstraction that is particularly tailored to *concurrent systems*. Finally, we introduce a number of enhancements of spotlight abstraction developed in this work. We will see that our enhanced approach to the abstraction generally enables us to efficiently verify concurrent systems on very small abstract models.

4.1 Predicate Abstraction

As we have discussed in the previous chapter, a state of a concurrent system corresponds to a valuation of all its variables. Hence, the state space of a system (i.e. the set of all states) is *exponential* in the number of system

variables. For real-life systems with large-domain variables we commonly obtain state spaces whose size widely exceeds the capabilities of state-of-the-art model checkers. This issue is generally known as the *state explosion problem* in verification. A well-established technique for reducing the state space complexity is *predicate abstraction* [66, 11]. In this approach a concrete system *Sys* over a set of variables *Var* is approximated by an *abstract system* Sys^a over a set of atomic predicates *Pred* (which are defined over *Var*). The benefit of predicate abstraction in verification is twofold. First, predicates can be restricted to the variables of particular interest in the verification task, and thus, *Pred* is usually significantly smaller than *Var*. And second, predicates commonly have a very limited domain: classical approaches are based on *boolean* abstractions, i.e. abstractions with a two-valued domain for predicates.

Applying predicate abstraction enables us to obtain abstract state spaces that are orders of magnitude smaller than the corresponding concrete ones. Therefore, we use this reduction technique as the basis of our approach to verification. Nevertheless, abstraction inherently involves a loss of information about the concrete system, and consequently, not every verification task can be successfully accomplished on a given abstraction. Abstractions that are too coarse for verification have to be *refined*, i.e the set of predicates has to be enlarged. Abstraction refinement will be the topic of Chapter 5.

In this section, we take a closer look at predicate abstraction. In particular, we show how the state space of our concurrent systems can be reduced, and how system properties can be preserved under this form of abstraction. We start with the classical *boolean predicate abstraction*.

4.1.1 Boolean Predicate Abstraction

Our work is based on the concept of boolean predicate abstraction [66, 11] (which we will extend to a three-valued domain in Section 4.1.2). Classical predicate abstraction-based verification tools (e.g. the model checkers Bebop [14] and BLAST [18]) transfer the *concrete* system under consideration into an *abstract* system where operations do not refer to concrete variables but to boolean predicates over these variables. In this section, we fundamentally introduce boolean predicate abstraction, following the approach of [11].

As a motivating example, we consider the concurrent system in Figure 4.1. Here we might be interested in verifying whether $Proc_2$ always terminates, or more formally, whether the CTL property $\mathbf{AF}(pc_2 = 3)$ holds. It is easy to see that in a fair computation the integer x will be eventually greater or equal to one, and thus, $Proc_2$ will finally execute the terminal operation *end*. However, following our formal approach to verification we first have to transfer the system into a state space model. Assuming that the domain of x and y is $\mathbb{Z}$, the set of reachable states is $S_{Var} = \{(x = 0, y = 0), (x = 1, y = 0), (x = 1, y =$

$-1),(x=2,y=-1),\ldots\}$, which means we obtain an infinite state space. Hence, a straightforward verification by exploring the *concrete* state space may fail due to the state explosion problem.

$$x,y : \textbf{integer where } x=0,y=0$$

$$Proc_1 :: \begin{bmatrix} 1: & \textbf{loop forever do} \\ \begin{bmatrix} 2: & x:=x+1 \\ 3: & y:=y-1 \end{bmatrix} & \end{bmatrix} \parallel Proc_2 :: \begin{bmatrix} 1: & \textbf{while } x<1 \textbf{ do} \\ \begin{bmatrix} 2: & \textbf{skip} \end{bmatrix} & \\ 3: & \textbf{end} \end{bmatrix}$$

Fig. 4.1 Concurrent system $Sys_7 = Proc_1 \parallel Proc_2$ over $Var = \{x,y\}$.

Now, the idea of predicate abstraction is to define a set of *boolean predicates* *Pred* over the system variables *Var* in order to obtain a much smaller (finite) state space. Thus, in abstract systems operations do not refer to concrete variables but to predicates, i.e. each concrete basic operation bop is approximated by an abstract operation bop_a with

$$bop_a \;\equiv\; assume(pe) : p_1 := pe_1,\ \ldots,\ p_k := pe_k$$

where $\{p_1,\ldots p_k\} = Pred$ and $pe, pe_1, \ldots, pe_k$ are boolean expressions over *Pred*. We generally assume that abstract operations assign to *all* predicates in *Pred*. Thus, an abstract operation bop_a that does not affect a predicate p_i contains the assignment $p_i := p_i$. In our examples we usually omit such self-assignments. Predicate expressions in abstract operations often take the form $choice(a,b)$ for boolean expressions a, b with the following semantics:

$$s(choice(a,b)) = \begin{cases} true & \text{if } s(a) \text{ is } true \\ false & \text{if } s(b) \text{ is } true \\ * & \text{else} \end{cases}$$

Here $*$ denotes a *non-deterministic* choice between *true* and *false*. Hence, applying boolean predicate abstraction may introduce non-deterministic guards and assignments.

For a *concrete* operation $bop \equiv assume(e) : x_1 := e_1,\ \ldots,\ x_m := e_m$ and a set of predicates $Pred = \{p_1,\ldots,p_k\}$ we can derive the corresponding *abstract* operation bop_a that approximates bop as follows. We start with the abstract assume condition pe:

$$pe \;\equiv\; choice(E_{Pred}(e), E_{Pred}(\neg e))$$

where $E_{Pred}(e)$ denotes the weakest boolean expression over *Pred* such that $E_{Pred}(e)$ semantically implies e, written $E_{Pred}(e) \models e$. Here 'weakest' means that all other boolean expressions pe over *Pred* with $pe \models e$, also imply $E_{Pred}(e)$.

Definition 4.1 (Weaker Boolean Expressions).
Let pe and pe' be boolean expressions with $pe \models pe'$ and $pe' \not\models pe$. Then pe' is *weaker* than pe.

As an example, for the set of predicates $Pred = \{(x < 0), (x < 1), (y < 0)\}$ and the expression $(x = 0)$ we get $E_{Pred}(x = 0) = \neg(x < 0) \wedge (x < 1)$. The expression $pe = \neg(x < 0) \wedge (x < 1) \wedge (y < 0)$ also semantically implies $(x = 0)$. However, we have that $pe \models E_{Pred}(x = 0)$ and $E_{Pred}(x = 0) \not\models pe$. Thus, $E_{Pred}(x = 0)$ is weaker than pe.

Furthermore, an abstract basic operation bop_a consists of an assignment $p_i := pe_i$ for each predicate $p_i \in Pred$ with:

$$pe_i \;\equiv\; choice(E_{Pred}(wp_{bop}(p_i)), E_{Pred}(wp_{bop}(\neg p_i)))$$

Here $wp_{bop}(p_i)$ is the *weakest precondition* of the concrete basic operation bop with regard to the predicate p_i.

Definition 4.2 (Weakest Precondition).
Let $bop \equiv assume(e) : x_1 := e_1, \ldots, x_m := e_m$ be a basic operation and p a predicate over the set of variables $Var = \{x_1, \ldots, x_m\}$. Then the *weakest precondition* of bop with respect to p is

$$wp_{bop}(p) = e \wedge p[x_1/e_1, \ldots, x_m/e_m]$$

where $p[x_1/e_1, \ldots, x_m/e_m]$ denotes the substitution of all occurrences of $x_1, \ldots, x_m$ in p by $e_1, \ldots, e_m$, respectively.

The notion of weakest preconditions was originally introduced by Dijkstra in the context of reasoning about imperative programs [52]. In terms of our systems, $wp_{bop}(p)$ corresponds to a predicate expression denoting the set of all states s over $Pred$ such that bop executed in s results in a state where p holds. E.g., the weakest precondition of $x := x + 1$ with regard to $(x < 1)$ is $wp_{x:=x+1}(x < 1) = (x < 0)$.

Now, we want to see how our example system Sys_7 can be abstracted over a set of predicates. First of all, we take a look at the *concrete* control flow graphs corresponding to the processes of Sys_7 (Figure 4.2).

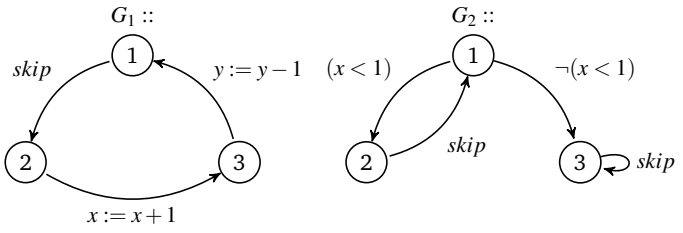

Fig. 4.2 Concrete control flow graphs G_1 and G_2 of the processes $Proc_1$ and $Proc_2$.

Here transitions are labelled with concrete basic operations over the variable set $Var = \{x, y\}$. For our abstraction we choose the predicate set $Pred = \{(x < 0), (x < 1)\}$. The corresponding *abstract* control flow graphs are depicted in Figure 4.3.

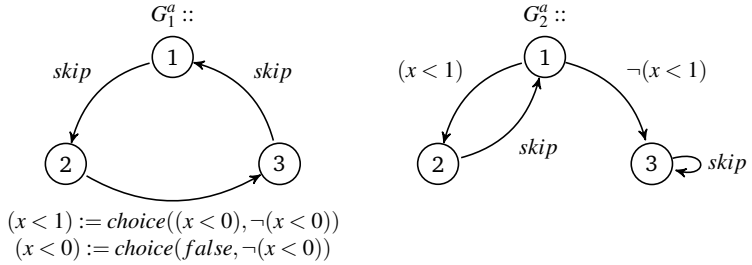

Fig. 4.3 Abstract control flow graphs G_1^a and G_2^a over $Pred = \{(x < 0), (x < 1)\}$.

These control flow graphs represent an abstraction Sys_7^a of the concrete system Sys_7 over the set of predicates $Pred = \{(x < 0), (x < 1)\}$. As we can see, the variable y is completely omitted in our abstraction; the concrete operation $y := y - 1$ is abstracted to *skip*. Furthermore, the operation $x := x + 1$ is abstracted to list of assignments with regard to the predicates $(x < 0)$ and $(x < 1)$. As the corresponding *abstract* state space we get

$$S_{Pred} = \{\big((x < 0), (x < 1)\big), \big(\neg(x < 0), (x < 1)\big), \big(\neg(x < 0), \neg(x < 1)\big)\}.$$

Alike concrete concurrent systems, abstract systems can be straightforwardly transferred into Kripke structures. We only have to take into account that such an *abstract* Kripke structure is now defined over S_{Pred} rather than over S_{Var}, and moreover, the transition relation R is defined with regard to abstract operations (see Definition 3.7). Model checking the CTL formula $\mathbf{AF}(pc_2 = 3)$ on a Kripke structure corresponding to Sys_7^a yields *true*. Hence, boolean predicate abstraction enables us to check temporal logic properties on very small abstractions.

However, so far we have no relation between model checking results obtained on concrete and abstract systems. We therefore define an *approximation relation* $\preceq$ for concrete systems Sys over Var and their abstractions Sys^a over $Pred$. We start on the level of boolean expressions:

Definition 4.3 (Approximation of Boolean Expressions).
Let Var be a set of variables and let e and e' be any two boolean expressions over Var. Then e *approximates* e', written $e \preceq e'$, iff

$$e \models e' \ \wedge \ \neg e \models \neg e'.$$

In our notion of abstract systems we have that *Pred* is a subset of the set of all boolean expressions over *Var*. Hence, we straightforwardly get that a predicate expression pe over *Pred* approximates a boolean expression e over *Var*, $pe \preceq e$, iff $pe \models e$ and $\neg pe \models \neg e$. In the particular case of *choice* expressions we have that $choice(a,b)$ approximates an expression e iff a semantically implies e and b semantically implies $\neg e$:

$$choice(a,b) \preceq e \quad \equiv \quad a \models e \ \wedge \ b \models \neg e$$

Next, we extend the approximation relation to basic operations:

Definition 4.4 (Approximation of Basic Operations).
Let *Var* be a set of variables and $Pred = \{p_1, \ldots, p_k\}$ a set of predicates over *Var*. Moreover, let $bop \equiv assume(e) : x_1 := e_1, \ldots, x_m := e_m$ be a concrete basic operation over *Var* and $bop_a \equiv assume(pe) : p_1 := pe_1, \ldots, p_k := pe_k$ an abstract basic operation over *Pred*. Then bop_a *approximates* bop, written $bop_a \preceq bop$, iff

$$pe \preceq e \ \wedge \ \bigwedge_{i=1}^{k} pe_i \preceq wp_{bop}(p_i).$$

Remember that an abstract basic operation generally assigns to *all* predicates. Hence, the approximation relation for basic operations is always defined with regard to a given set of predicates *Pred*. If *Pred* is clear from the context, we will not explicitly mention it.

For illustration, we again consider our running example and show that $(x < 0) := choice(false, \neg(x < 0))$ approximates $x := x + 1$, which is equivalent to

$$\begin{aligned}
& choice(false, \neg(x<0)) \ \preceq \ wp_{x:=x+1}(x<0) \\
\equiv \ & choice(false, \neg(x<0)) \ \preceq \ (x<-1) \\
\equiv \ & false \models (x<-1) \ \wedge \ \neg(x<0) \models \neg(x<-1) \\
\equiv \ & true.
\end{aligned}$$

Predicate abstraction solely affects basic operations – and *not* the control structure of processes. Hence, a process $Proc_i^a$ approximates a process $Proc_i$ iff they have isomorphic control flow graphs and the basic operations in $Proc_i^a$ approximate the corresponding ones in $Proc_i$.[1] Finally, a concurrent system $Sys^a = \|_{i=1}^n Proc_i^a$ approximates a system $Sys = \|_{i=1}^n Proc_i$ iff each process $Proc_i^a$ in Sys^a approximates the corresponding process $Proc_i$ in Sys. The subsequent definition gives us a formal notion of the approximation relation for concurrent systems:

[1] Remember that abstract operations bop_a may introduce non-determinism, and thus, an abstract control flow transition $\delta_i^a(l_i, bop_a, l_i')$ may be equivalent to *two* transitions between the same locations but labelled with different operations. This, however, does not affect isomorphism in our notion.

Definition 4.5 (Approximation of Concurrent Systems).
Let $Sys = \|_{i=1}^{n} Proc_i$ be a concrete concurrent system over Var and let $Sys^a = \|_{i=1}^{n} Proc_i^a$ be an abstract concurrent system over $Pred$. Moreover, let $G_1, \ldots, G_n$, respectively $G_1^a, \ldots, G_n^a$ be the control flow graphs of the processes $Proc_1, \ldots, Proc_n$, resp. $Proc_1^a, \ldots, Proc_n^a$. Then Sys^a approximates Sys, written $Sys^a \preceq Sys$, iff for all $i = 1, \ldots, n$: G_i and G_i^a are *isomorphic*, i.e. there exists a bijective function $h : Loc_i \to Loc_i^a$, also called *isomorphism*, with

- $\forall l \in Loc_i : h(l) = l$,
  (*which actually implies that* $Loc_i = Loc_i^a$)
- $\delta_i(l, bop, l')$ if there exists a bop_a with $bop_a \preceq bop$ and $\delta_i^a(h(l), bop_a, h(l'))$,
- $\delta_i^a(h(l), bop_a, h(l'))$ if there exists a bop with $bop_a \preceq bop$ and $\delta_i(l, bop, l')$.

As we can see, the requirement that each basic operation in the concrete system is approximated by the corresponding one in the abstract system is already included in our notion of isomorphism.

Coming back to our running example, we get by Definition 4.5 that our abstract system Sys_7^a properly approximates the concrete system Sys_7. In general, applying boolean predicate abstraction to a concurrent system Sys gives us a *conservative over-approximation* of Sys. This means an abstraction Sys_a may permit *more* possible behaviour than the original Sys, i.e. all computations that are feasible in Sys are also feasible in Sys_a – but a feasible computation of Sys_a is not necessarily feasible in Sys. Transferred to the field of temporal logic model checking, we get that if a property from the universal fragment of CTL holds for the abstract system then we can deduce that it holds for the corresponding concrete system as well. From [11] we can derive Theorem 4.1 which relates model checking results of concrete and abstract systems for *corresponding states*.

Definition 4.6 (Corresponding States in Boolean Abstractions).
Let $Sys = \|_{i=1}^{n} Proc_i$ be a concurrent system over Var, and $Sys^a = \|_{i=1}^{n} Proc_i^a$ an abstract system over $Pred$, with $Sys^a \preceq Sys$. Moreover, let $K = (S, R, L, \mathbb{F})$ be the Kripke structure representing Sys, and $K^a = (S^a, R^a, L^a, \mathbb{F}^a)$ the Kripke structure representing Sys^a where K and K^a are both defined over the same set of atomic predicates AP. Then a concrete state $s \in S$ *corresponds* to an abstract state $s^a \in S^a$ iff

$$\forall p \in AP : L(s, p) = L^a(s^a, p).$$

Thus, corresponding states in boolean abstractions have the same labelling with regard to AP, which we sometimes abbreviate by $L(s) = L^a(s^a)$.

Theorem 4.1.
Let $Sys = \|_{i=1}^{n} Proc_i$ be a concurrent system over Var, and $Sys^a = \|_{i=1}^{n} Proc_i^a$ an abstract system over Pred, with $Sys^a \preceq Sys$. Moreover, let $K = (S, R, L, \mathbb{F})$ be the

Kripke structure representing Sys, and $K^a = (S^a, R^a, L^a, \mathbb{F}^a)$ *the Kripke structure representing* Sys^a *where K and* K^a *are both defined over the same set of atomic predicates AP. Then for any two corresponding states* $s \in S$ *and* $s^a \in S^a$ *and for any ACTL formula* ψ *over control locations or the predicates in Pred the following holds:*

$$[K^a, s^a \models \psi] \quad \Rightarrow \quad [K, s \models \psi]$$

Proof (Theorem 4.1).
We prove this theorem by showing that the notion of corresponding states in boolean abstractions (Definition 4.6) conforms to a fair simulation (Definition 2.7). This can be established based on the following corollary which we get from [11]:

Corollary 4.1.
Let $Sys = \|_{i=1}^n Proc_i$ *be a concurrent system over Var, and* $Sys^a = \|_{i=1}^n Proc_i^a$ *an abstract system over Pred, with* $Sys^a \preceq Sys$. *Moreover, let* $K = (S, R, L, \mathbb{F})$ *be the Kripke structure representing Sys, and* $K^a = (S^a, R^a, L^a, \mathbb{F}^a)$ *the Kripke structure representing* Sys^a *where K and* K^a *are both defined over the same set of atomic predicates AP. Let* $s \in S$ *and* $s^a \in S^a$ *be any two corresponding states. Then for every path* $\pi \in \Pi_s$ *in K there is a path* $\pi^a \in \Pi_{s^a}$ *in* K^a *such that for all* $k \in \mathbb{N} : L(\pi_k) = L^a(\pi_k^a)$, *i.e* π_k *and* π_k^a *are corresponding states.*

Hence, for every path in K there is a corresponding path in K^a or, rather, for any two states $s \in S$ and $s^a \in S^a$ with $L(s) = L^a(s^a)$ we have that:

$$R(s, s') \quad \Rightarrow \quad \exists s'^a \in S^a : R^a(s^a, s'^a) \ \wedge \ L(s') = L^a(s'^a)$$

We have to strengthen this result with regard to fairness, i.e. we require that for every *fair* path in K there is a corresponding *fair* path in K^a. This follows by induction from Lemma 4.1.

Lemma 4.1.
Let $Sys = \|_{i=1}^n Proc_i$ *be a concurrent system over Var, and* $Sys^a = \|_{i=1}^n Proc_i^a$ *an abstract system over Pred, with* $Sys^a \preceq Sys$. *Moreover, let* $K = (S, R, L, \mathbb{F})$ *be the Kripke structure representing Sys, and* $K^a = (S^a, R^a, L^a, \mathbb{F}^a)$ *the Kripke structure representing* Sys^a *where K and* K^a *are both defined over the same set of atomic predicates AP. Let* $s \in S$ *and* $s^a \in S^a$ *be any two states with* $L(s) = L^a(s^a)$. *Furthermore, let* s' *be any state in S and* $i \in [1..n]$ *a process index then*

$$R_i(s, s') \quad \Rightarrow \quad \exists s'^a \in S^a : R_i^a(s^a, s'^a) \ \wedge \ L(s') = L^a(s'^a)$$

(Remember that, according to Definition 3.7, the transition function R can be reformulated as $\bigvee_{i=1}^n R_i$.*)*

Proof (Lemma 4.1).
$R_i(s, s')$ can be rewritten into $R_i(\langle l, s_{Var} \rangle, \langle l', s'_{Var} \rangle)$ where l denotes the location part and s_{Var} the variable part of the state s. $R_i(\langle l, s_{Var} \rangle, \langle l', s'_{Var} \rangle)$ refers to a

control flow transition $\delta_i(l, bop, l')$ in $G_i = (Loc_i, \delta_i)$. According to Definition 4.5 there exists a corresponding transition $\delta_i^a(l, bop_a, l')$ in G_i^a with $bop_a \preceq bop$. By assumption, we have that bop can be executed in s and $L(s) = L^a(s^a)$. Hence, bop_a can be executed in s^a, i.e. for some state $s'^a \in S^a$ there exists a transition $R_i(s^a, s'^a)$ associated with bop_a. Since $bop_a \preceq bop$ we have that $L(s') = L^a(s'^a)$. □

From Lemma 4.1 we can deduce that our notion of corresponding states in boolean abstractions (Definition 4.6) conforms to a *fair simulation* (see Definition 2.7).

Corollary 4.2.
Let $Sys = \|_{i=1}^n Proc_i$ be a concurrent system over Var, and $Sys^a = \|_{i=1}^n Proc_i^a$ an abstract system over Pred, with $Sys^a \preceq Sys$. Moreover, let $K = (S, R, L, \mathbb{F})$ be the Kripke structure representing Sys, and $K^a = (S^a, R^a, L^a, \mathbb{F}^a)$ the Kripke structure representing Sys^a where K and K^a are both defined over the same set of atomic predicates AP. Then the set of all pairs of corresponding states $s \in S$ and $s^a \in S^a$ characterises a fair simulation $\preceq_s S \times S^a$ between K and K^a.

The correctness of Theorem 4.1 now immediately follows from Theorem 2.2, which states that fair simulation preserves ACTL properties. □

Hence, our result that the property $\mathbf{AF}(pc_2 = 3)$ holds for the small abstraction Sys_7^a can be directly transferred to the concrete system Sys_7. In general, boolean predicate abstraction enables us to verify concurrent systems on very small abstract models, and thus, helps us to reduce the complexity of temporal logic model checking.

However, if checking a universally quantified property yields *false* for an abstraction, we can not conclude that the original system violates this property as well. In this case, the model checking procedure additionally returns an *abstract counterexample* – a path in the abstract model that refutes the property. In order to gain certainty about whether this counterexample is *spurious* (i.e. it exists in the abstraction only) or corresponds to a *real* path, it has to be retraced on the original system. The *retracement* of counterexamples involves a partial exploration of the concrete state space, and thus, suffers from the state explosion problem as well. In the next section we introduce *three-valued* predicate abstraction. This generalisation of boolean predicate abstraction is capable of preserving both *true* and *false* results in verification – which saves us the expensive retracement step. Moreover, three-valued predicate abstraction is straightforwardly compatible with spotlight abstraction – the second abstraction technique that we employ in our framework (compare Section 4.2).

4.1.2 Three-Valued Predicate Abstraction

Applying abstraction to software systems always means, that some details about the original system get lost, or rather, become unknown. Thus, an intuitive way of characterising this loss of information in an abstract model is to introduce a third truth value *unknown* for predicates and transitions. Such a *three-valued* sematics for state space models was first proposed in [22], and since then has been used in many abstraction frameworks (e.g. [115, 4, 87]). Three-valued abstractions moreover have the advantage that *true* as well as *false* results in verification can be preserved.

In our approach to the verification of concurrent systems, we also employ a three-valued abstraction. Similar to the boolean abstraction, we have predicates instead of concrete variables, and thus, abstract basic operations again take the form

$$bop_a \;\equiv\; assume(pe) : p_1 := pe_1, \ldots, p_k := pe_k.$$

But now, our predicates have a three-valued domain: the valuation of a predicate in an abstract state can be *true*, *false* or *unknown*. Unknown is in fact a valid truth value as we operate with the Kleene logic $\mathbb{K}_3$ [62] (see Section 2.2). Three-valued expressions in abstract operations may again take the form $choice(a,b)$, but now with the following semantics:

$$s(choice(a,b)) = \begin{cases} true & \text{if } s(a) \text{ is } true \\ false & \text{if } s(b) \text{ is } true \\ \bot & \text{else} \end{cases}$$

Hence, non-deterministic choices are resolved in three-valued abstractions. Instead, the third truth value *unknown* may be assigned to predicates, or may occur as a guard.

The derivation of a *three-valued* abstract system Sys^a for a given concrete system Sys and a set of predicates $Pred$ can be directly adopted from the boolean predicate abstraction (see Section 4.1.1). We only have to consider that predicate expressions are now evaluated under the Kleene logic, and that we have a new semantics for *choice*. Furthermore, the approximation relation $\preceq$ for boolean abstractions can be straightforwardly extended to the three-valued setting. Three-valued abstractions naturally require three-valued state space models. As shown in Section 2.2, classical Kripke structures can be generalised to *three-valued Kripke structures*, which are tailored to model partially unknown systems. Using three-valued Kripke structures moreover gives rise to a new notion of correspondence between concrete and abstract states.

Definition 4.7 (Corresponding States in Three-Valued Abstractions).
Let $Sys = \|_{i=1}^{n} Proc_i$ be a concurrent system over Var, and $Sys^a = \|_{i=1}^{n} Proc_i^a$

a three-valued abstraction over *Pred*, with $Sys^a \preceq Sys$. Moreover, let $K = (S, R, L, \mathbb{F})$ be the Kripke structure representing *Sys*, and $K^a = (S^a, R^a, L^a, \mathbb{F}^a)$ the three-valued Kripke structure representing Sys^a where K and K^a are both defined over the same set of atomic predicates AP. Then a concrete state $s \in S$ *corresponds* to an abstract state $s^a \in S^a$ iff

$$\forall p \in AP : L^a(s^a, p) \leq_{\mathbb{K}_3} L(s, p)$$

Thus, if a predicate $p \in AP$ has a definite value (*true*, *false*) in an abstract state s^a, then p has the same value in a corresponding concrete state s. However, predicates that are valuated with *unknown* in s^a may be *true* or *false* in s. We also say, the labelling of the concrete state s is *more definite* than the labelling of a corresponding abstract state s^a, abbreviated by $L^a(s^a) \leq_{\mathbb{K}_3} L(s)$.

Applying three-valued predicate abstraction gives us a conservative approximation in the sense that all definite behaviour in the abstract system is also feasible in the concrete one. Hence, *all* definite verification results, i.e. *true* and *false*, obtained on the abstraction can be directly transferred to the original system. Only an *unknown* result tells us nothing about the concrete system. From [112] we get the following theorem:

Theorem 4.2.
Let $Sys = \|_{i=1}^{n} Proc_i$ be a concurrent system over Var, and $Sys^a = \|_{i=1}^{n} Proc_i^a$ a three-valued abstraction over Pred, with $Sys^a \preceq Sys$. Moreover, let $K = (S, R, L, \mathbb{F})$ be the Kripke structure representing Sys, and $K^a = (S^a, R^a, L^a, \mathbb{F}^a)$ the three-valued Kripke structure representing Sys^a where K and K^a are both defined over the same set of atomic predicates AP. Then for any two corresponding states $s \in S$ and $s^a \in S^a$ and for any CTL formula ψ over control locations or the predicates in Pred the following holds:

$$[K^a, s^a \models \psi] \ \leq_{\mathbb{K}_3} \ [K, s \models \psi]$$

To illustrate the consequence of this theorem for our approach to verification, we take a look at the concurrent system Sys_8 in Figure 4.4.

$$x, y : \textbf{integer where } x = 1, y = 1$$

$$Proc_1 :: \begin{bmatrix} 1: & \textbf{loop forever do} \\ \lceil 2: & x := -x \rceil \end{bmatrix} \parallel Proc_2 :: \begin{bmatrix} 1: & \textbf{while } x > 0 \textbf{ do} \\ \lceil 2: & y := y - 1 \rceil \\ 3: & \textbf{end} \end{bmatrix}$$

Fig. 4.4 Concurrent system $Sys_8 = Proc_1 \parallel Proc_2$ over $Var = \{x, y\}$.

We want to verify whether $Proc_2$ always eventually terminates, i.e. whether the CTL formula $\mathbf{AF}(pc_2 = 3)$ holds for the concurrent system. It is easy to see that this property is violated for Sys_8; e.g. consider a computation where $Proc_1$ executes the loop body always twice in succession before $Proc_2$

evaluates its *while* condition. However, for the purpose of *formal* verification we first construct a *three-valued abstraction* of our system over the set of predicates $Pred = \{(x > 0), (x > -1), (y > 0)\}$. Now, model checking the CTL formula on our abstract system Sys_8^a yields *false*, and additionally returns a counterexample. According to Theorem 4.2 we can straightforwardly deduce that the original system violates $\mathbf{AF}(pc_2 = 3)$ as well. Moreover, we have the guarantee that the revealed counterexample corresponds to a real path in Sys_8, and thus, no additional retracement step is required. – In contrast, for an analogous *boolean abstraction* (over the same set of predicates) model checking also returns *false* together with a counterexample; but this result has to validated by retracing the counterexample on the original system.

Naturally, for a fixed set of predicates a three-valued abstraction is always slightly larger than a boolean abstraction because now we have a three-valued (instead of a two-valued) domain for predicates. Nevertheless, for many real-life verification tasks it is more advantageous to have a slightly larger abstract model than to be forced to perform an additional retracement step. Remember that retracing counterexamples involves a partial exploration of the concrete state space, and thus, might suffer from the state explosion problem.

Three-valued abstractions give us more precision in temporal logic model checking because both *true* and *false* results are preserved. However, verifying a three-valued abstraction might also yield *unknown*, which tells us nothing about the concrete system. We e.g. get such a result when checking the CTL formula $\mathbf{AG}(y > 0)$ on our aforementioned abstraction (over $Pred = \{(x > 0), (x > -1), (y > 0)\}$). An *unknown* result always comes along with an *unconfirmed counterexample* – a potential error path in the abstract system with some unknown transitions and predicates. Now, we can directly conclude that our abstraction is too coarse for a definite result in verification. The abstraction can then be *refined* based on an analysis of the unconfirmed counterexample (see Chapter 5). – Contrary, verifying the formula $\mathbf{AG}(y > 0)$ on an analogous boolean abstraction returns *false* together with an *abstract* counterexample. An additional retracement step is required, which reveals that the counterexample is *spurious* and that the abstraction is too coarse.

So far, we have seen that (boolean and three-valued) predicate abstraction is a powerful technique in cutting down the state space of software systems. In particular, predicate abstraction enables us to restrict the large (or even infinite) domains of system variables to very small fragments, and thus, can improve the efficiency of formal verification by orders of magnitude. However, in *concurrent* systems the space complexity furthermore exponentially grows with the number of processes composed in parallel. In the next section, we introduce *spotlight abstraction* – a specific abstraction technique for concurrent systems that can be combined with predicate abstraction.

4.2 Spotlight Abstraction

The state space complexity of concurrent systems does not only depend on the quantity of system variables but also on the processes composed in parallel. Remember that a computation of a concurrent system corresponds to a sequence of non-deterministic selections of processes that are permitted to execute their next operation. Hence, each additional process increases the number of possible computations – and so the number of reachable states – *exponentially*.

Predicate abstraction allows us to cope with state explosion caused by large-domain variables. But we still suffer from the state space complexity induced by *concurrency*. However, we will see that predicate abstraction can be combined with *spotlight abstraction* – a reduction technique tailored to concurrent systems. The spotlight principle was initially introduced by Wachter and Westphal [123] in the context of parameterised verification. Later it was enriched with three-valued CTL semantics and integrated into a framework for abstraction refinement [112]. The underlying idea of spotlight abstraction is to set a *spotlight* on certain processes of particular interest while the remaining ones are kept in the *shade*. Now, the spotlight processes are thoroughly considered when constructing the abstract system, whereas processes in the shade are nearly completely abstracted away.

Applying predicate abstraction *combined* with spotlight abstraction for concurrent systems enables us to tackle *both*, large-domain variables and large numbers of processes. Again, an abstract system might be too coarse for a definite result in verification, and thus, has to be refined. The spotlight principle also adds a new facet to abstraction refinement because now we can choose between adding a new predicate or a process from the shade (see Chapter 5).

In this section, we give a detailed introduction to the spotlight principle. In particular, we show how spotlight abstraction can be integrated into our approach to the verification of concurrent systems. Moreover, we introduce certain extensions of the spotlight principle that allow us to preserve more concrete behaviour in the abstraction, without an additional growth of the state space.

4.2.1 Spotlight and Shade

The spotlight principle is based on the idea to divide a concurrent system into a *spotlight* and a *shade*, or rather, into processes of particular interest and processes that are presumably not relevant for the underlying verification task. In order to gain a better understanding of what 'interesting' and 'presumably not relevant' may mean in the context of concurrent system verification, we consider the message passing system in Figure 4.5.

$$c : \textbf{channel}\ [1]\ \textbf{of integer}$$

$$Proc_1 :: \begin{bmatrix} \textbf{local}\ x : \textbf{integer where}\ x = 0 \\ 1: \quad \textbf{loop forever do} \\ \quad \begin{bmatrix} 2: \quad \textbf{receive}(c,x) \\ 3: \quad \textbf{progress} \end{bmatrix} \end{bmatrix} \parallel Proc_2 :: \begin{bmatrix} 1: \quad \textbf{loop forever do} \\ \quad \begin{bmatrix} 2: \quad \textbf{send}(c,1) \\ 3: \quad \textbf{progress} \end{bmatrix} \end{bmatrix}$$

$$\parallel_{i=3}^{n} Proc_i :: \begin{bmatrix} \textbf{local}\ y_i : \textbf{integer where}\ y_i = 0 \\ 1: \quad \textbf{loop forever do} \\ \quad \begin{bmatrix} 2: \quad \textbf{receive}(c,y_i) \\ 3: \quad \textbf{progress} \end{bmatrix} \end{bmatrix}$$

Fig. 4.5 Message passing system $Sys_9 = \|_{i=1}^{n} Proc_i$ over $Var = \{c, x, y_3, \ldots, y_n\}$.

Here we have n processes communicating via the channel c. $Proc_1$ continuously attempts to *receive* a value from c, whereas $Proc_2$ continuously attempts to *send* a value to the channel. All other processes also attempt to *receive* on c. The *progress* in the loop bodies is synonymous for the empty operation. Now, we are interested in verifying whether there is continuous *progress* for $Proc_1$, or more formally, whether the liveness property $\mathbf{AG}(\mathbf{AF}(pc_1 = 3))$ holds for Sys_9. It is evident that $Proc_1$ is *of interest* in this verification task because exactly *this* process is referenced in our temporal logic formula. However, the progress of $Proc_1$ depends on the availability of communication partners, or respectively, on the presence of competitors. In fact, it is sufficient to take a detailed look at the process of interest $Proc_1$, the potential communication partner $Proc_2$ and one competitor e.g. $Proc_3$ in order to refute the formula. The argument here is that there exists a fair computation where $Proc_2$ and $Proc_3$ successfully execute their communication operations in turns, whereas $Proc_1$ solely attempts to receive when the channel is empty. Hence, we have a computation where $Proc_1$ starves at location 2. In particular, we see that the processes $Proc_4$ to $Proc_n$ are *not relevant* here.

Spotlight abstraction allows us to exploit the fact that usually only *some* processes of a concurrent systems are relevant for a given verification task. The general approach is to apply classical three-valued predicate abstraction to the processes in the spotlight whereas shade processes are summarised into one approximative component $Proc_{Shade}$. This process entirely neglects the original control flow of the processes in the shade. – Instead it approximates operations on shared variables occurring in shade processes by continuously modifying predicates over those variables. Due to the approximative character of $Proc_{Shade}$ and the inherent loss of information about the shade processes, predicates might be set to *unknown*.

To illustrate the spotlight principle, we look again at the message passing system Sys_9. We take the processes $Proc_1$ to $Proc_3$ into the spotlight and construct their three-valued abstractions $Proc_1^a$, $Proc_2^a$ and $Proc_3^a$ over the predicates $empty_c$ and $full_c$. Thus, the processes $Proc_4$ to $Proc_n$ are in the shade and we summarise them into the approximative component $Proc_{Shade}$, which is depicted in Figure 4.6.

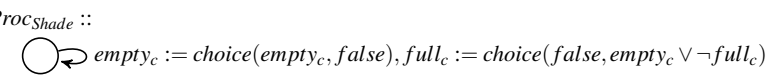

Fig. 4.6 Control flow representation of $Proc_{Shade}$.

As we can see, $Proc_{Shade}$ solely consists of a single control flow location with a self-loop. The associated abstract operation approximates *all* concrete basic operations with regard to $empty_c$ and $full_c$ that occur in the shade processes. Remember that in our notion of message passing systems all shared variables are channel-related; which means, a bop occurring in a shade process $Proc_i$ is either a channel operation or an operation that solely modifies variables local to $Proc_i$. The latter case implies that bop is not affecting any processes in the spotlight. Thus, in our current example $Proc_{Shade}$ only refers to the channel-related predicates $empty_c$ and $full_c$. For general concurrent systems $Proc_{Shade}$ might also refer to predicates over ordinary shared variables.

Now, our overall abstract system Sys_9^a corresponds to the parallel composition of $Proc_1^a$, $Proc_2^a$, $Proc_3^a$ and $Proc_{Shade}$. Hence, Sys_9^a is significantly smaller than an analogous *pure* predicate abstraction where $Proc_4$ to $Proc_n$ are considered in detail. Model checking the CTL formula $\mathbf{AG}(\mathbf{AF}(pc_1 = 3))$ on our small spotlight abstraction Sys_9^a yields *false*. In the following, we show that also spotlight abstraction preserves *true* and *false* properties, and hence, that we can transfer our verification result to the concrete system.

First of all, we provide some fundamental definitions with regard to the spotlight principle. Let $Sys = \|_{i=1}^n Proc_i$ be a concurrent system over a set of variables Var. Then we denote the set of all processes of Sys by $Proc = \{Proc_1, \ldots, Proc_n\}$. Moreover, we denote the set of all predicates over Var by $Pred$. Note that, unlike before, $Pred$ now refers to *all* possible predicates over the system variables, and thus, this set is generally infinitely large. Now, a *spotlight abstraction* of a concurrent system Sys is defined by a set of *spotlight processes* $Spot(Proc) \subseteq Proc$, and a set of *spotlight predicates* $Spot(Pred) \subseteq Pred$. Consequently, approximations are henceforth defined with regard to the finite set of predicates $Spot(Pred)$. We refer to the overall spotlight as $Spot = Spot(Proc) \cup Spot(Pred)$. The corresponding shade is characterised by the complementary sets $Shade(Proc) = Proc \setminus Spot(Proc)$ and $Shade(Pred) = Pred \setminus Spot(Pred)$. For the overall shade we get $Shade = Shade(Proc) \cup Shade(Pred)$. Hence, a spotlight abstraction of a concurrent system can also be regarded as a partition of the system's processes and predicates into the sets $Spot$ and $Shade$. Figure 4.7 illustrates the aforementioned set relations based on our running example. Three-valued spotlight abstraction now can be applied to concurrent systems according to the following definition:

Definition 4.8 (Spotlight Abstraction of Concurrent Systems).
Let $Sys = \|_{i=1}^n Proc_i$ be a concurrent system and $Pred$ the set of all predicates over the system variables. Moreover, let $Spot = Spot(Proc) \cup Spot(Pred)$ be a

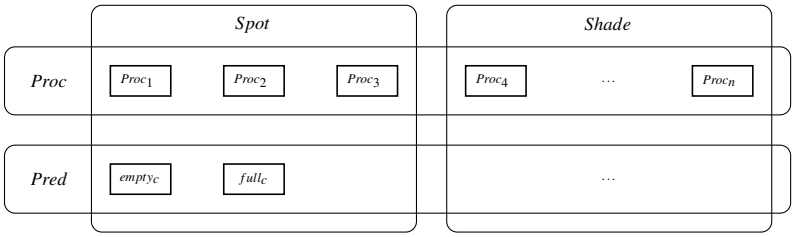

Fig. 4.7 Spotlight abstraction of Sys_9 given by a partition of the system's processes and predicates into *Spot* and *Shade*.

given set of spotlight processes and predicates, and let $Shade = Shade(Proc) \cup Shade(Pred)$ be the corresponding shade. Then the abstract system $Sys^a = \|_{Proc_i \in Spot(Proc)} Proc_i^a \parallel Proc_{Shade}$ approximates Sys iff

- for every $Proc_i \in Spot(Proc)$: $Proc_i^a$ approximates $Proc_i$.
- $Proc_{Shade}$ is a control flow graph with one location and a single loop labelled with an abstract basic operation bop_{Shade} over $Spot(Pred)$ such that *all* concrete basic operations bop that occur in shade processes are approximated by bop_{Shade}.

This is the basic definition of three-valued spotlight abstraction, taken with slight changes from [112]. In the original approach every spotlight predicate that is modified in the shade is just set to *unknown* in bop_{Shade} – which is the most generalised, and thus, coarsest form of an abstract operation that approximates all concrete operations occurring in the shade. – Our weaker requirements to $Proc_{Shade}$ still guarantee correctness (see Corollary 4.3) and additionally permit us to preserve more definite behaviour in the abstraction. For illustration, we consider again our running example Sys_9 with $Spot(Proc) = \{Proc_1, Proc_2\}$, $Spot(Pred) = \{empty_c, full_c\}$ and the temporal logic formula $\mathbf{AG}(\mathbf{AF}(pc_1 = 3))$. Now, according to the original approach [112] the shade component $Proc_{Shade}$ would set $empty_c$ and $full_c$ continuously to *unknown*. This native abstraction is not precise enough to give us a definite answer in our verification task. – In comparison, our shade component in Figure 4.6 still conforms to Definition 4.8 and moreover gives us the formerly missing precision. Within this work, we have developed a number of further optimisations of the shade component that enable us to preserve more concrete behaviour (at the same abstraction size), and thus, to obtain more definite results in verification. We discuss these enhancements separately in the subsequent sections (Section 4.2.2 and 4.2.3).

From a theorem of [112] we obtain the following corollary, which relates the verification results of concrete and abstract systems:

Corollary 4.3.
Let $Sys = \|_{i=1}^{n} Proc_i$ be a concurrent system and Pred the set of all predicates

over the system variables. Moreover, let $Spot = Spot(Proc) \cup Spot(Pred)$ *be a given set of spotlight processes and predicates, and* $Sys^a = \|_{Proc_i \in Spot(Proc)} Proc_i^a \parallel Proc_{Shade}$ *the corresponding abstract system with* $Sys^a \preceq Sys$*. Furthermore, let* $K = (S, R, L, \mathbb{F})$ *be the Kripke structure representing Sys, and* $K^a = (S^a, R^a, L^a, \mathbb{F}^a)$ *the three-valued Kripke structure representing* Sys^a*, both defined over* $AP = Spot(Pred) \cup \{pc_i = j | Proc_i \in Spot(Proc), j \in Loc_i\}$*. Then for any two corresponding states* $s \in S$ *and* $s^a \in S^a$ *and for any CTL formula* ψ *over AP the following holds:*

$$[K^a, s^a \models \psi] \ \leq_{\mathbb{K}_3} \ [K, s \models \psi]$$

The original proof in [112] relies on the following argument: bop_{Shade} sets *all* spotlight predicates to *unknown*, and thus, approximates *all* operations occurring in the shade. In our slightly modified approach, bop_{Shade}, *by definition*, approximates *all* operations in the shade, but is not necessarily as abstract as the native bop_{Shade}. This relaxation, however, does not affect the argumentation of the proof in [112].

Applying spotlight abstraction enables us to check temporal logic properties of concurrent systems on usually very small abstract models. Similar to pure three-valued predicate abstraction, definite results can be transferred to the original systems. But now, we can additionally abstract away complete processes that are presumably not relevant for the underlying verification task. Even if a process is summarised in the shade, the information about its behaviour is not entirely lost. For illustration, we look again at our running example. Here we have a number of *receiver* processes in the shade. Thus, the predicates $empty_c$ and $full_c$ might be modified by $Proc_{Shade}$ in an unknown manner. However, according to the semantics of communication channels, a *receive* applied to an empty channel has no effect, i.e. the channel will definitely remain *empty*. Exactly this fact is incorporated in our shade component (Figure 4.6) by the assignment $empty_c := choice(empty_c, false)$. We see, that specific knowledge about the system (in our example, the semantics of *receive*) may let us preserve more concrete behaviour in the shade, and thus, may give us more definite results in verification. In the following two sections, we introduce further enhancements of the shade that are also based on the exploitation of easily accessible information about the considered system.

4.2.2 Shade Clustering

The spotlight principle lets us abstract away entire processes of concurrent systems, and thus, gives us very small models for verification. It is based on the simple but usually very effective idea of setting predicates that are modified by processes in the shade to *unknown*. However, this kind of abstraction may

remove more information about the considered system than actually necessary. For illustration, we look at the message passing system in Figure 4.8.

$$c : \textbf{channel}\ [1]\ \textbf{of integer}$$

$$Proc_1 :: \begin{bmatrix} 1: & \textbf{loop forever do} \\ \begin{bmatrix} 2: & \textbf{send}(c,1) \end{bmatrix} & \\ \begin{bmatrix} 3: & \textbf{progress} \end{bmatrix} & \end{bmatrix} \parallel Proc_2 :: \begin{bmatrix} \textbf{local}\ x : \textbf{integer where}\ x = 0 \\ 1: \quad \textbf{loop forever do} \\ \begin{bmatrix} 2: & \textbf{receive}(c,x) \end{bmatrix} \end{bmatrix}$$

$$\|_{i=3}^{n}\ Proc_i :: \begin{bmatrix} \textbf{local}\ y_i : \textbf{integer where}\ y_i = 0 \\ 1: \quad \textbf{loop forever do} \\ \begin{bmatrix} 2: & \textbf{receive}(c,y_i) \end{bmatrix} \end{bmatrix}$$

Fig. 4.8 Message passing system $Sys_{10} = \|_{i=1}^{n}\ Proc_i$ over $Var = \{c, x, y_3, \ldots, y_n\}$.

Here we have one sender process $Proc_1$ and a number of receiver processes $Proc_2$ and $Proc_3$ to $Proc_n$. All these processes communicate via the channel c. We want to validate that $Proc_1$ continuously reaches its *progress* location, i.e. $\mathbf{AG}(\mathbf{AF}(pc_1 = 3))$. Intuitively, it is sufficient to look at the sender $Proc_1$ and at one receiver e.g. $Proc_2$; and moreover, to regard the fact that there is no other sender in the system. For every *send* by $Proc_1$ there will be a *receive* by $Proc_2$ (or by other receiver processes). Hence, the channel c will be empty infinitely often and we can conclude that $Proc_1$ always eventually proceeds. However, setting the spotlight on $Proc_1$ and $Proc_2$, and accordingly, keeping $Proc_3$ to $Proc_n$ in the shade, would not give us this definite answer in our approach to verification. The problem here is that there will be always an execution where the shade component sets the predicate $full_c$ to *unknown*. But this predicate is crucial for the progress of $Proc_1$; and hence, we will get no definite result unless the shade is completely empty. In general, validating universally quantified properties which are dependent on the result or success of a communication operation requires every potential communication partner and competitor to be in the spotlight. Due to transitive dependencies caused by message passing this often leads to an entirely empty shade, and thus, compromises the positive effect of abstraction.

In this section, we introduce an extended approach to spotlight abstraction for message passing systems which bypasses the aforementioned problem. Instead of summarising the shade processes into a *single* component we now use a *set* of shade components, each abstracting the communication between the spotlight and the shade on a distinct channel. Such a *shade clustering* can be applied to message passing systems according to the following definition.

Definition 4.9 (Shade Clustering for Message Passing Systems).
Let $Sys = \|_{i=1}^{n}\ Proc_i$ be a message passing system and $Pred$ the set of all predicates over the system variables. Moreover, let $Spot = Spot(Proc) \cup Spot(Pred)$ be a given set of spotlight processes and predicates, and let $Shade = Shade(Proc) \cup Shade(Pred)$ be the corresponding shade. Then the abstract system $Sys^a = \|_{Proc_i \in Spot(Proc)}\ Proc_i^a \parallel Proc_{Shade}$ approximates Sys iff

- for every $Proc_i \in Spot(Proc)$: $Proc_i^a$ approximates $Proc_i$ with respect to $Spot(Pred)$.

- $Proc_{Shade} = \|_{j=0}^{m} Proc_{Shade}^{j}$ is a parallel composition such that

  1. $Proc_{Shade}^{0}$ is a control flow graph with one location and a single loop labelled with $bop_a \equiv assume(\bot)$, i.e. an *unknown* transition guard and an empty assignment part.

  2. if there exists a channel c that is referenced in $Spot(Pred)$ and there occurs a *send* operation on this channel c in some process in $Shade(Proc)$, then we have a $Proc_{Shade}^{j}$ which is a CFG with one location and a single loop labelled with $bop_a \equiv assume(\bot) : bop'_a$ such that the assignment part bop'_a approximates all assignment parts of successful[2] *send* operations on c that occur in $Shade(Proc)$.

  3. if there exists a channel c that is referenced in $Spot(Pred)$ and there occurs a *receive* operation on this channel c in some process in $Shade(Proc)$, then we have a $Proc_{Shade}^{j}$ which is a CFG with one location and a single loop labelled with $bop_a \equiv assume(\bot) : bop'_a$ such that the assignment part bop'_a approximates all assignment parts of successful *receive* operations on c that occur in $Shade(Proc)$.

  4. $Proc_{Shade} = \|_{j=0}^{m} Proc_{Shade}^{j}$ is solely composed of CFGs according to 1, 2 and 3. In case that there are no CFGs according to 2 and 3, then there exists no $Proc_{Shade}$ in the abstract system.

This extended approach to spotlight abstraction is based on the following observation: if there exists a communication operation e.g. $receive(c, y_i)$ in some process $Proc_i$ in the shade, then it is *either* currently enabled (with regard to $Proc_i$'s local control flow) and can be executed *or* not. In spotlight abstraction with shade clustering we again omit the original control flow of shade processes. Instead, for each channel c and each type of operation *send/receive* that is relevant for the communication between the spotlight and the shade, we model the possible branches (e.g. either successful *receive* or nothing happens) as loop transitions – each with the assume condition *unknown* – of single-node CFGs. The cases where nothing happens are clustered in the shade component $Proc_{Shade}^{0}$, whereas for the successful execution of a communication operation on a particular channel we have a distinct shade component $Proc_{Shade}^{j}$. Now, for our example system Sys_{10} with $Shade(Proc) = \{Proc_3, \ldots, Proc_n\}$, $Shade(Pred) = \{empty_c, full_c\}$ the shade components look as depicted in Figure 4.9.

[2] Remember that *send* is a compound operation consisting of two basic operations: A *successful send* in the case that the channel is not full, and an *unsuccessful send* which causes busy waiting.

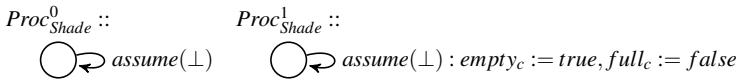

Fig. 4.9 Control flow representation of the shade components $Proc^0_{Shade}$ and $Proc^1_{Shade}$.

As we can see, $Proc^0_{Shade}$ clusters all cases where some shade process executes an operation that has no impact on the spotlight (e.g. an attempt to receive on an empty channel). – Conversely, $Proc^1_{Shade}$ clusters all successful executions of *receive* operations by shade processes. With our shade clustering, we moreover get an extended notion of fairness. In a fair computation of an abstract system Sys^a *some* shade component (but not necessarily *all* shade components) has to proceed infinitely often. For temporal logic model checking we now can exploit the fact that several CTL properties with regard to spotlight processes are preserved under *all* possible branches provided by the shade components. In our example, we e.g. do not know whether the next execution step of the shade will be a successful *receive* or an operation that does not affect the spotlight. However, under *both* of these branches $Proc_1$ will continuously be able to execute *send*. Thus, checking $\mathbf{AG}(\mathbf{AF}(pc_1 = 3))$ on our spotlight abstraction with shade clustering yields *true*; and according to Theorem 4.3 we can transfer this result to the original system.

Theorem 4.3.
Let $Sys = \|_{i=1}^{n} Proc_i$ be a message passing system and Pred the set of all predicates over the system variables. Moreover, let $Spot = Spot(Proc) \cup Spot(Pred)$ be a given set of spotlight processes and predicates, and $Sys^a = \|_{Proc_i \in Spot(Proc)} Proc_i^a \| Proc_{Shade}$ the corresponding abstract system with shade clustering and $Sys^a \preceq Sys$. Furthermore, let $K = (S, R, L, \mathbb{F})$ be the Kripke structure representing Sys, and $K^a = (S^a, R^a, L^a, \mathbb{F}^a)$ the three-valued Kripke structure representing Sys^a, both defined over $AP = Spot(Pred) \cup \{pc_i = j | Proc_i \in Spot(Proc), j \in Loc_i\}$. Then for any two corresponding states $s \in S$ and $s^a \in S^a$ and for any CTL formula ψ over AP the following holds:

$$[K^a, s^a \models \psi] \ \leq_{\mathbb{K}_3} \ [K, s \models \psi]$$

Proof (Theorem 4.3).
We prove this theorem by showing that the notion of corresponding states in three-valued abstractions (Definition 4.7) with shade clustering (Definition 4.9) conforms to a fair concreteness preorder (Definition 2.12). This can be established based on the following lemma:

Lemma 4.2.
Let $Sys = \|_{i=1}^{n} Proc_i$ be a message passing system and Pred the set of all predicates over the system variables. Moreover, let $Spot = Spot(Proc) \cup Spot(Pred)$ be a given set of spotlight processes and predicates, and $Sys^a = \|_{Proc_i \in Spot(Proc)} Proc_i^a \|$

$Proc_{Shade}$ the corresponding abstract system with shade clustering and $Sys^a \preceq Sys$. Furthermore, let $K = (S, R, L, \mathbb{F})$ be the Kripke structure representing Sys, and $K^a = (S^a, R^a, L^a, \mathbb{F}^a)$ the three-valued Kripke structure representing Sys^a, both defined over $AP = Spot(Pred) \cup \{pc_i = j | Proc_i \in Spot(Proc), j \in Loc_i\}$. Let $s \in S$ and $s^a \in S^a$ be any two corresponding states, i.e. $L^a(s^a) \leq_{\mathbb{K}_3} L(s)$. Then we have that:

1. *For every $Proc_i \in Spot(Proc)$:*

   a. *If $s'^a \in S^a$ is any abstract state such that $R_i^a(s^a, s'^a) = true$, then there is a concrete state $s' \in S$ such that $R_i(s, s') = true$ and $L^a(s'^a) \leq_{\mathbb{K}_3} L(s')$.*

   b. *If $s' \in S$ is any concrete state such that $R_i(s, s') \neq false$, then there is an abstract state $s'^a \in S^a$ such that $R_i^a(s^a, s'^a) \neq false$ and $L^a(s'^a) \leq_{\mathbb{K}_3} L(s')$.*

2. *For every $Proc_i \in Shade(Proc)$:*

   a. *If $s'^a \in S^a$ is any abstract state such that $R_{Shade}^a(s^a, s'^a) = true$, then there is a concrete state $s' \in S$ such that $R_i(s, s') = true$ and $L^a(s'^a) \leq_{\mathbb{K}_3} L(s')$.*

   b. *If $s' \in S$ is any concrete state such that $R_i(s, s') \neq false$, then there is an abstract state $s'^a \in S^a$ such that $R_{Shade}^a(s^a, s'^a) \neq false$ and $L^a(s'^a) \leq_{\mathbb{K}_3} L(s')$.*

Where R_i refers to a transition associated with $Proc_i$, and R_{Shade}^a refers to a transition associated with some shade component.

Proof (Lemma 4.2).
Case 1 was proven by Schrieb et al. in [112]. Our shade clustering solely concerns the second case. 2 (a) is trivially fulfilled because for all abstract transitions associated with the shade we have that $R_{Shade}^a(s^a, s'^a) = \bot$, and thus, the premise of 2 (a) never holds. The proof of 2 (b) goes as follows: Let bop be the concrete basic operation associated with $R_i(s, s')$. We now can distinguish three cases:

1. bop does not affect any predicates in $Spot(Pred)$ (i.e. bop does not modify any variables that are referenced in some $p \in Spot(Pred)$). Then we select $bop_a \equiv assume(\bot)$ as the corresponding abstract operation (See Definition 4.9 (1)). Note that the assignment part of bop_a is empty – this, however, approximates the assignment part of op with respect to $Spot(Pred)$.

2. bop affects some predicates in $Spot(Pred)$ and bop corresponds to the successful execution of a *send* operation on some channel c. Then we select $bop_a \equiv assume(\bot) : bop'_a$ as the corresponding abstract operation where bop'_a is defined according to Definition 4.9 (2).

3. bop affects some predicates in $Spot(Pred)$ and bop corresponds to the successful execution of a *receive* operation on some channel c. Then we select $bop_a \equiv assume(\bot) : bop'_a$ as the corresponding abstract operation where bop'_a is defined according to Definition 4.9 (3).

According to the definition of shade clustering, we have that for each type of concrete basic operation bop occurring in $Proc(Shade)$, there exists a shade component $Proc^j_{Shade}$ with a self-loop labelled with the corresponding abstract operation bop_a. Hence, this abstract operation bop_a can be executed in every abstract state. Moreover, bop_a always approximates bop with respect to $Spot(Pred)$: By Definition 4.9 the abstract assignment part bop'_a approximates the concrete assignment part bop'; the abstract guard is always $assume(\bot)$, which approximates all possible concrete guards. Now, let s'^a be the abstract state resulting from the execution of bop_a in s^a. We already have the premise that $L^a(s^a) \leq_{\mathbb{K}_3} L(s)$, thus, we can conclude that $L^a(s^a) \leq_{\mathbb{K}_3} L(s)$ holds as well. Furthermore, by definition of the shade we have $R^a_{Shade}(s^a, s'^a) = \bot$ (due to the guards $assume(\bot)$), and thus, $R^a_{Shade}(s^a, s'^a) \neq false$. □

From this lemma we get the following corollary by induction:

Corollary 4.4.
Let $Sys = \|^n_{i=1} Proc_i$ be a message passing system and $Pred$ the set of all predicates over the system variables. Moreover, let $Spot = Spot(Proc) \cup Spot(Pred)$ be a given set of spotlight processes and predicates, and $Sys^a = \|_{Proc_i \in Spot(Proc)} Proc^a_i \| Proc_{Shade}$ the corresponding abstract system with an shade clustering and $Sys^a \preceq Sys$. Furthermore, let $K = (S, R, L, \mathbb{F})$ be the Kripke structure representing Sys, and $K^a = (S^a, R^a, L^a, \mathbb{F}^a)$ the three-valued Kripke structure representing Sys^a, both defined over $AP = Spot(Pred) \cup \{pc_i = j | Proc_i \in Spot(Proc), j \in Loc_i\}$. Let $s \in S$ and $s^a \in S^a$ be any two corresponding states, i.e. $L^a(s^a) \leq_{\mathbb{K}_3} L(s)$. Then we have that:

1. *For every fair path $\pi^a \in \Pi^{fair}_{s^a}$ in K^a exists a fair path $\pi \in \Pi^{fair}_s$ in K with $\forall k \in \mathbb{N}_{>0}$:*

$$R^a(\pi^a_{k-1}, \pi^a_k) = true \;\Rightarrow\; R(\pi_{k-1}, \pi_k) = true \wedge L^a(\pi^a_k) \leq L(\pi_k)$$

2. *For every fair path $\pi \in \Pi^{fair}_s$ in K exists a fair path $\pi^a \in \Pi^{fair}_{s^a}$ in K^a with $\forall k \in \mathbb{N}_{>0}$:*

$$R(\pi_{k-1}, \pi_k) \neq false \;\Rightarrow\; R^a(\pi^a_{k-1}, \pi^a_k) \neq false \wedge L^a(\pi^a_k) \leq L(\pi_k)$$

This result together with Definition 2.12 gives us another corollary:

Corollary 4.5.
Let $Sys = \|^n_{i=1} Proc_i$ be a message passing system and $Pred$ the set of all predicates over the system variables. Moreover, let $Spot = Spot(Proc) \cup Spot(Pred)$ be a given set of spotlight processes and predicates, and $Sys^a = \|_{Proc_i \in Spot(Proc)} Proc^a_i \|$

$Proc_{Shade}$ the corresponding abstract system with shade clustering and $Sys^a \preceq Sys$. Furthermore, let $K = (S, R, L, \mathbb{F})$ be the Kripke structure representing Sys, and $K^a = (S^a, R^a, L^a, \mathbb{F}^a)$ the three-valued Kripke structure representing Sys^a, both defined over $AP = Spot(Pred) \cup \{pc_i = j | Proc_i \in Spot(Proc), j \in Loc_i\}$. Then the set of all pairs of corresponding states $s \in S$ and $s^a \in S^a$ characterises a fair concreteness preorder $\preceq_c S^a \times S$ between K^a and K.

The correctness of Theorem 4.3 now immediately follows from Theorem 2.3, which states that a fair concreteness preorder preserves all CTL properties. □

Spotlight abstraction with shade clustering for message passing systems enables us to preserve some concrete behaviour of shade processes that gets lost in classical spotlight abstraction. More precisely, we can exploit the fact that in message passing systems processes only communicate in a restricted way (i.e either by *send* or by *receive*). The general idea of shade clustering is to model all the possible behaviour of the shade processes as distinct 'unknown' branches. Then, temporal logic properties that are preserved under all of these branches also hold for the concrete system. The clustering of the shade may lead to a slightly larger abstraction, since we now have a shade component for each possible branch. However, this extended abstraction can give us definite verification results in cases where the classical approach requires *all* processes to be in the spotlight. In the next section we will see, that the preservation of more concrete behaviour is not only possible by modifying the shade, but also by modifying the spotlight.

4.2.3 Region Summarisation

Spotlight abstraction is based on a three-valued domain for predicates. The third truth value *unknown* is then propagated to the level of Krikpe structures and temporal logic formulae. A three-valued Kripke structure is used to model the abstract, and thus, partially unknown system. Based on the three-valued CTL semantics (Definition 2.10) it can be deduced whether such a model is still concrete enough to obtain a definite result in verification. This approach is known as three-valued model checking [22, 23] – a verification technique that can be generally applied to any partially unknown system that is representable as a three-valued Kripke structure. However, the universality of three-valued model checking also entails that certain details about the system that exceed the modelling capabilities of Kripke structures, can not be regarded in the verification task. This may lead to cases where, according to the three-valued CTL semantics, model checking yields *unknown* – whereas an additional look at the semantics of the system would clearly reveal a definite result. In this section, we show that additional knowledge about the considered system can

sometimes be exploited in constructing the abstraction, which finally gives us more definite results in verification.

At first, we take a look at a typical verification task where we obtain *unknown* under the three-valued CTL semantics, but a short glance at the system already reveals a definite answer. In Figure 4.10 we have a message passing system where several processes communicate via channel c, but only $Proc_1$ attempts to communicate via d.

$$c,d : \textbf{channel}\ [1]\ \textbf{of integer}$$

$$Proc_1 :: \begin{bmatrix} 1: & \textbf{receive}(c,x) \\ 2: & \textbf{receive}(d,y) \\ 3: & \textbf{end} \end{bmatrix} \|_{i=2}^{n} Proc_i :: \begin{bmatrix} \dots \\ \textbf{send}(c,1) \\ \dots \end{bmatrix} \|_{j=n+1}^{m} Proc_j :: \begin{bmatrix} \dots \\ \textbf{receive}(c,z_j) \\ \dots \end{bmatrix}$$

Fig. 4.10 Message passing system $Sys_{11} = \|_{i=1}^{n} Proc_i$ over $Var = \{c,d,x,y,z_{n+1},\dots,z_m\}$.

Obviously, $Proc_1$ will never terminate because no communication partner for channel d is available. Following our formal approach to verification, we check the temporal logic property $\mathbf{AF}(pc_1 = 3)$. The model checker internally negates this universally quantified formula, i.e. it verifies whether the negation $\mathbf{EG}\neg(pc_1 = 3)$ does *not* hold for the system. Now, we construct a spotlight abstraction of the system, given by $Spot(Proc) = \{Proc_1\}$ and $Spot(Pred) = \{empty_d\}$. Note that we require no shade component here, because the shade processes $Proc_2$ to $Proc_m$ do not affect the spotlight predicate $empty_d$. The resulting abstract control flow graph of $Proc_1$ and the corresponding three-valued Kripke structure are depicted in Figure 4.11.

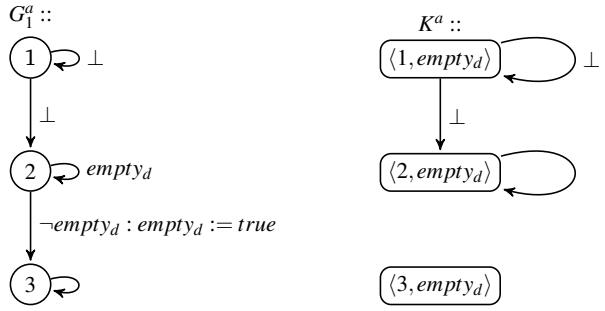

Fig. 4.11 Abstract control flow graph G_1^a of $Proc_1$ over $Spot(Pred) = \{empty_d\}$ and corresponding three-valued Kripke structure K^a.

When we look at the Kripke structure, it is still obvious that there exists no path starting in the state $\langle 1, empty_d \rangle$ that reaches the *end* location. However, all paths have at least one transition with the truth value $\bot$. Hence, checking $\mathbf{EG}\neg(pc_1 = 3)$ yields *unknown* according to ther three-valued CTL semantics.

And consequently, neither for $\mathbf{EG}\neg(pc_1 = 3)$ nor for $\mathbf{AF}(pc_1 = 3)$ we obtain a definite verification result on the current abstraction.

Now, we will see how this abstraction can be enriched by additional knowledge about the concurrent system Sys_{11}. Considering $Proc_1$ in Figure 4.10, we observe that the control flow from location 1 to 2 depends on the success of the operation $receive(c,x)$. More precisely, an *empty* channel c will cause busy waiting at location 1, whereas a *non-empty* channel permits the process to proceed to location 2. The corresponding control flow transitions $1 \to 1$ and $1 \to 2$ have *complementary* guards. Hence, in every state where $Proc_1$ is at location 1, *either* a transition associated with $1 \to 1$ is enabled and a transition associated with $1 \to 2$ is disabled, *or* vice versa. We want to exploit this fact for our abstraction, without adding the predicate $empty_c$ to the spotlight. Therefore, we look again at the three-valued Kripke structure K^a in Figure 4.11. According to the concreteness preorder (see Definition 2.11), we have that each CTL formulae that evaluates to a definite value on K^a, evaluates to the same truth value on every concretisation of K^a. Such a concretisation can be obtained by replacing the '*unknowns*' in the three-valued Kripke structure by *true* or *false*. Considering the fact that we have identified a pair of complementary transitions in the abstraction, we get exactly the following two-valued concretisations of K^a:

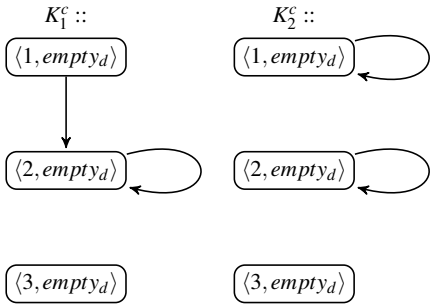

Fig. 4.12 Two-valued concretisations K_1^c and K_2^c of the three-valued Kripke structure K^a.

As we can see, the temporal logic formula $\mathbf{AF}(pc_1 = 3)$ holds for *none* of the concretisations of K^a. The fact that K_1^c and K_2^c are the only possible two-valued concretisations allows us to transfer this refutation result to the three-valued Kripke structure K^a, and finally, also to the concrete system Sys_{11}. However, incorporating this exemplified approach into a model checking procedure is not straightforward. The construction of *all* two-valued concretisations of a three-valued Kripke structure generally involves an exponential overhead – which is usually unacceptable for the verification of real-life systems.

Nevertheless, we will see that we can exploit the basic idea of constructing concretisations in our approach to abstraction – which eventually gives us more precision *without* increasing the complexity. Looking again at Figure

4.12, we can make the observation that in any case (or rather, under each concretisation) the system will remain in the states $\langle 1, empty_d \rangle$ and $\langle 2, empty_d \rangle$ forever – but it is still unpredictable, in which *exact* state the system will be at a certain point of time. This missing detail is however not relevant in the current verification task. In our extended approach to abstraction, we introduce *regions* of states: subsets of the overall state space where we are solely interested in whether the system will remain *inside* or *outside* the set. Our running example thus hints at the states $\langle 1, empty_d \rangle$ and $\langle 2, empty_d \rangle$ as a region. We will see that by defining a region, we can subsume a *set* of possible concretisations in *one* region state. In our technique we detect the regions on the level of control flow graphs and subsequently propagate them to the states of the corresponding Kripke structure. Hence, we choose the locations 1 and 2 of the process $Proc_1$ as our region. In the next step, we perform *region summarisation* – a local modification of the abstraction.

Definition 4.10 (Region Summarisation).
Let $Sys = \|_{i=1}^{n} Proc_i$ be a concurrent system and $Pred$ the set of all predicates over the system variables. Let $Spot = Spot(Proc) \cup Spot(Pred)$ be a given set of spotlight processes and predicates. Moreover, let $Proc_k \in Spot(Proc)$ be a spotlight process with the corresponding control flow graph $G = (Loc, \delta)$, and let $Reg \subseteq Loc$ be a subset of G's locations, called a *region*. Then $G_{Reg} = (Loc_{Reg}, \delta_{Reg})$ approximates G iff

- $Loc_{Reg} := (Loc \backslash Reg) \cup \{r\}$.
  (*The locations in the region Reg are replaced by the new location r.*)

- δ_{Reg} :

  1. Let $\delta(l, bop, l')$ with $l \in Loc \backslash Reg$ and $l' \in Loc \backslash Reg$. Then $\delta_{Reg}(l, bop_a, l')$ with $bop_a \preceq bop$.
     (*Transitions outside the region are not affected by the summarisation.*)

  2. Let $\delta(l, bop, l')$ with $l \in Loc \backslash Reg$ and $l' \in Reg$. Then $\delta_{Reg}(l, bop_a, r)$ with $bop_a \preceq bop$.
     (*Transitions entering the region are redirected to the new location r.*)

  3. Let $\delta(l, bop, l')$ with $l \in Reg$, $l' \in Loc \backslash Reg$ and $bop = grd : ass$ where grd denotes the guard of bop and ass is the assignment part of bop. Then $\delta_{Reg}(r, grd_a \wedge \bot : ass_a, l')$ with $grd_a \preceq grd$ and $ass_a \preceq ass$.
     (*Transitions leaving the region now start at the new location r and their guards are conjuncted with* $\bot$.)

  4. Let Grd be the set of guards on transitions from Reg to $Loc \backslash Reg$ in G and let Ass be the set of assignments on transitions within Reg. Then $\delta_{Reg}(r, grd_a : ass_a, r)$ with $grd_a \equiv \bot \vee \bigwedge_{g \in Grd} \neg g_a$ where $g_a \preceq g$, and $\forall ass \in Ass : ass_a \preceq ass$.

(*Transitions within the region are summarised into one self-loop of r, the new guard is* $\bot$*, disjuncted with a conjunction over all negated guards of transitions leaving the region. The new assignment approximates all assignments on transitions within Reg.*)

Each transition in G_{Reg} conforms to either $1, 2, 3$ or 4.

According to this definition, we can summarise the region (i.e. the subset of locations) $Reg = \{1,2\}$ of the spotlight process $Proc_1$ into a new location r. This modification requires an update of the transition relation of the corresponding control flow graph. For the region location r we introduce a self-loop labelled with an abstract basic operation $grd_a : ass_a$ (4). Since we want to map all original transitions *within* the region *Reg* onto this new self-loop, the assignment part ass_a has to approximate all assignments of basic operations associated with these original transitions. In our example, the transitions within *Reg* have no assignments, and thus, there is no assignment part in our $grd_a : ass_a$. The only way to leave a region is to take a transition starting in *Reg* that leads to a location outside the region. Moreover, the guard of this transition has to be *true*. Hence, as long as all guards associated with such outgoing transitions are *false*, the process will definitely remain in the region. As a consequence, the guard of the new self-loop is a conjunction over all negated guards of the transitions leaving the region – which is just $empty_d$ in our example. Even if one of the guards of transitions leaving the region is *true*, there is no guarantee that the process will leave the region – the process will *maybe* still remain inside. Therefore, we disjunct the guard of the self-loop with $\bot$. Next, we look at transitions that in fact leave the region (3). After applying region summarisation, these transitions start in the new location r whereas their destination location lies outside the region. Since we have lost the information about the exact location within the region, it is uncertain whether these transitions can be actually taken, and thus, we conjunct their guards with $\bot$. Figure 4.13 illustrates the application of region summarisation for our running example, and moreover, shows the corresponding Kripke structure.

As we can see, in the finally obtained Kripke structure the former states $\langle 1, empty_d \rangle$ and $\langle 2, empty_d \rangle$ are summarised into the region state $\langle r, empty_d \rangle$. These state unites all possible concretisations (compare Figure 4.12) of our original Kripke structure (compare Figure 4.11). The price that we pay is the loss of information about the exact control flow location of $Proc_1$ within the region $Reg = \{1,2\}$. This is however not relevant for verifying $\mathbf{AF}(pc_1 = 3)$. Model checking this temporal logic property yields *false* for our new Kripke structure, and again we can transfer this result to the concrete system:

Theorem 4.4.
Let $Sys = \|_{i=1}^{n} Proc_i$ *be a concurrent system and* $Spot = Spot(Proc) \cup Spot(Pred)$ *be a given set of spotlight processes and predicates. Moreover, let* $Proc_k \in Spot(Proc)$ *be a spotlight process and let Reg be a region of its control locations. Let* $Sys^a = \|_{Proc_i \in Spot(Proc)} Proc_i^a \parallel Proc_{Shade}$ *be the corresponding abstract*

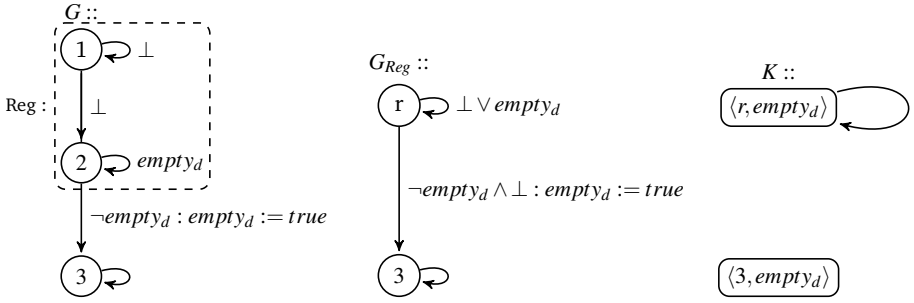

Fig. 4.13 Abstract control flow graph G of $Proc_1$ over $Spot(Pred) = \{empty_d\}$ with selected region $Reg = \{1,2\}$. Control flow graph G_{Reg} after applying region summarisation. Corresponding Kripke structure K.

system where region summarisation is applied to $Proc_k$ with regard to Reg. Furthermore, let $K = (S,R,L,\mathbb{F})$ be the three-valued Kripke structure representing Sys, and $K^a = (S^a,R^a,L^a,\mathbb{F}^a)$ the three-valued Kripke structure representing Sys^a, both defined over $AP = Spot(Pred) \cup \{pc_i = j | Proc_i \in Spot(Proc) \setminus Proc_k, j \in Loc_i\} \cup \{pc_k = j | j \in Loc_k \setminus Reg \vee j = r\}$. Then for any two corresponding states $s \in S$ and $s^a \in S^a$, i.e. $L^a(s^a) \leq_{\mathbb{K}_3} L(s)$, and for any CTL formula ψ over AP the following holds:

$$[K^a, s^a \models \psi] \ \leq_{\mathbb{K}_3} \ [K, s \models \psi]$$

Note that the Kripke structures are no longer defined over control locations inside the region.

Proof (Theorem 4.4).
We prove this theorem by showing that the notion of corresponding states in three-valued abstractions (Definition 4.7) with region summarisation (Definition 4.10) conforms to a fair concreteness preorder (Definition 2.12). This can be established based on the following lemma:

Lemma 4.3.
Let K and K^a be defined as in Theorem 4.4. Moreover, let $s \in S$ and $s^a \in S^a$ be any two corresponding states, i.e. $L^a(s^a) \leq_{\mathbb{K}_3} L(s)$. Then for the spotlight process $Proc_k$ with the summarised region Reg we have that:

1. *If $s'^a \in S^a$ is any abstract state such that $R^a_k(s^a, s'^a) = true$, then there is a concrete state $s' \in S$ such that $R_k(s,s') = true$ and $L^a(s'^a) \leq_{\mathbb{K}_3} L(s')$.*

2. *If $s' \in S$ is any concrete state such that $R_k(s,s') \neq false$, then there is an abstract state $s'^a \in S^a$ such that $R^a_k(s^a, s'^a) \neq false$ and $L^a(s'^a) \leq_{\mathbb{K}_3} L(s')$.*

Proof (Lemma 4.3).
We start with (1). Let $G = (Loc, \delta)$ be the CFG of $Proc_k$ *before* applying region summarisation, and let $G_{Reg} = (Loc_{Reg}, \delta_{Reg})$ be the abstract CFG of $Proc_k$ *after*

applying region summarisation. Moreover, let l be the location of G_{Reg} in s^a and l' be the location of G_{Reg} in s'^a. We can distinguish the following cases:

- The transition $R_k^a(s^a, s'^a)$ is independent from the region Reg. Then $l \neq r$ and $l' \neq r$, and there is an abstract basic operation bop_a with $\delta_{Reg}(l, bop_a, l')$. According to Definition 4.10 (1) there exists a transition $\delta(l, bop, l')$ in G with $bop_a \preceq bop$. Thus, taking the transition corresponding to $\delta(l, bop, l')$ in the concrete state s of K leads us to a state s' with $L^a(s'^a) \leq_{\mathbb{K}_3} L(s')$.

- $R_k^a(s^a, s'^a)$ is a transition that enters the region Reg. Then $l \neq r$ and $l' = r$, and there is an abstract basic operation bop_a with $\delta_{Reg}(l, bop_a, l')$. According to Definition 4.10 (2) there exists a location $\tilde{l} \in Reg$ with $\delta(l, bop, \tilde{l})$ in G and $bop_a \preceq bop$. Thus, taking the transition corresponding to $\delta(l, bop, \tilde{l})$ in the concrete state s of K leads us to a state s' with $L^a(s'^a) \leq_{\mathbb{K}_3} L(s')$.

- $R_k^a(s^a, s'^a)$ is a transition that leaves the region Reg. Then $l = r$ and $l' \neq r$, and there is an abstract basic operation bop_a with $\delta_{Reg}(l, bop_a, l')$. According to Definition 4.10 (3) the guard of bop_a contains a conjunction with $\bot$. We can conclude that $R_k^a(s^a, s'^a) <_{\mathbb{K}_3} true$, i.e. it is not a definite transition. Hence, the premise of 1 does not hold.

- $R_k^a(s^a, s'^a)$ corresponds to the self-loop associated with region Reg. Then $l = r$ and $l' = r$, and there is an abstract basic operation $bop_a = grd_a : ass_a$ with $\delta_{Reg}(l, bop_a, l')$. The premise of 1 is that $R_k^a(s^a, s'^a) = true$. According to Definition 4.10 (4), this is only the case if s^a is a state where *all* guards of transitions in G that leave the region evaluate to *false*. According to the semantics of our systems, in each state there is at least one enabled transition for each process. Hence, there must be two locations $\hat{l}, \check{l}$ in Reg with $\delta(\hat{l}, grd : ass, \check{l})$ in G and the guard grd is *true* in s. According to Definition 4.10 (4), the assignment part ass_a of the operation bop_a in $\delta_{Reg}(l, bop_a, l')$ approximates *all* assignments of operations on transitions within Reg, and thus, we have that $ass_a \preceq ass$. Hence, taking the transition corresponding to $\delta(\hat{l}, grd : ass, \check{l})$ in the concrete state s of K leads us to a state s' with $L^a(s'^a) \leq_{\mathbb{K}_3} L(s')$.

Next, we prove (2). Let $G = (Loc, \delta)$ be the CFG of $Proc_k$ *before* applying region summarisation, and let $G_{Reg} = (Loc_{Reg}, \delta_{Reg})$ be the abstract CFG of $Proc_k$ *after* applying region summarisation. Moreover, let l be the location of G in s and l' be the location of G in s'. We again can distinguish the following cases:

- $R_k(s, s')$ is a transition independent from the region Reg. Then $l \notin Reg$ and $l' \notin Reg$, and there is a basic operation bop with $\delta(l, bop, l')$. According to Definition 4.10 (1), there exists a transition $\delta_{Reg}(l, bop_a, l')$ in G_{Reg} with $bop_a \preceq bop$. Thus, taking the transition corresponding to $\delta_{Reg}(l, bop_a, l')$ in the abstract state s^a of K^a leads us to a state s'^a with $L^a(s'^a) \leq_{\mathbb{K}_3} L(s')$.

- $R_k(s,s')$ is a transition that enters the region Reg. Then $l \notin Reg$ and $l' \in Reg$, and there is a basic operation bop with $\delta(l,bop,l')$. According to Definition 4.10 (2), there exists a transition $\delta_{Reg}(l,bop_a,r)$ in G_{Reg} with $bop_a \preceq bop$. Thus, taking the transition corresponding to $\delta_{Reg}(l,bop_a,r)$ in the abstract state s^a leads us to a state s'^a with $L^a(s'^a) \leq_{\mathbb{K}_3} L(s')$.

- $R_k(s,s')$ is a transition that leaves the region Reg. Then $l \in Reg$ and $l' \notin Reg$, and there is a basic operation $bop = grd : ass$ with $\delta(l,bop,l')$. According to Definition 4.10 (3), there exists a transition $\delta_{Reg}(r,bop_a,l)$ in G_{Reg} with $bop_a = grd_a \wedge \bot : ass_a$ where $grd_a \preceq grd$ and $ass_a \preceq ass$; i.e. bop_a approximates bop. Since $l \in Reg$, we have that $L^a(s^a, pc_j = r) = true$. Thus, taking the transition corresponding to $\delta_{Reg}(r,bop_a,l)$ in the abstract state s^a leads us to a state s'^a with $L^a(s'^a) \leq_{\mathbb{K}_3} L(s')$.

- $R_k(s,s')$ is a transition within the region Reg. Then $l \in Reg$ and $l' \in Reg$, and there is a basic operation $bop = grd : ass$ with $\delta(l,bop,l')$. According to Definition 4.10 (4), there is a transition $\delta_{Reg}(r,grd_a : ass_a,r)$ in G_{Reg} with $grd_a \equiv \bot \vee \bigwedge_{g \in Grd} \neg g_a$ where $g_a \preceq g$ and $\forall ass \in Ass : ass_a \preceq ass$. Hence, ass_a approximates the assignment part ass of bop. Moreover, we have that $R^a_k(s^a,s'^a) \neq false$ because the guard grd_a contains a disjunction with $\bot$. Thus, taking the transition corresponding to $\delta_{Reg}(r,grd_a : ass_a,r)$ in s^a leads us to a state s'^a with $L^a(s'^a) \leq_{\mathbb{K}_3} L(s')$.

□

In Theorem 4.4 we assume that region summarisation is applied to a single process $Proc_k$ only. Transitions associated with processes different to $Proc_k$ are not affected by our modification of the abstraction. Hence, from Lemma 4.3 together with our prior results on spotlight abstraction we get the following corollary:

Corollary 4.6.
Let $Sys = \|_{i=1}^{n} Proc_i$ be a concurrent system and $Spot = Spot(Proc) \cup Spot(Pred)$ be a given set of spotlight processes and predicates. Moreover, let $Proc_k \in Spot(Proc)$ be a spotlight process and let Reg be a region of its control locations. Let $Sys^a = \|_{Proc_i \in Spot(Proc)} Proc_i^a \parallel Proc_{Shade}$ be the corresponding abstract system where region summarisation is applied to $Proc_k$ with regard to Reg. Furthermore, let $K = (S,R,L,\mathbb{F})$ be the three-valued Kripke structure representing Sys, and $K^a = (S^a,R^a,L^a,\mathbb{F}^a)$ the three-valued Kripke structure representing Sys^a, both defined over $AP = Spot(Pred) \cup \{pc_i = j | Proc_i \in Spot(Proc) \setminus Proc_k, j \in Loc_i\} \cup \{pc_k = j | j \in Loc_k \setminus Reg \vee j = r\}$. Then the set of all pairs of corresponding states $s \in S$ and $s^a \in S^a$ characterises a fair concreteness preorder $\preceq_c S^a \times S$ between K^a and K.

The correctness of Theorem 4.4 now immediately follows from Theorem 2.3, which states that a fair concreteness preorder preserves all CTL properties.
□

Region summarisation is a technique that allows us to modify three-valued abstractions based on additional knowledge about the considered system. In certain cases this gives us more definite results in verification – specifically, in cases where we want to validate that in all computations sets of particular control locations are never left. Since applying region summarisation principally means to sum up parts of a process' control flow in one location, this technique does not entail any further growth of the state space. It rather counteracts state explosion in terms of reducing the number of control locations, but still can give us more definite verification results. Region summarisation corresponds to a local modification of the abstraction, i.e. building a region affects the control flow of a single process. Although we have only considered the single application of region summarisation, this technique can be applied several times in one abstraction, i.e. multiple regions can be constructed for different processes. It remains the question of how to select expedient regions in a fully-automatic approach to verification, or more precisely, how to integrate region summarisation into our overall framework. We already have seen that the set possible concretisations of a three-valued model hints at a region, but the construction of all these concretisations usually causes an unacceptable computational overhead. In the next chapter, we will see how imprecise abstractions of concurrent systems can be iteratively refined under heuristic guidance, which also includes an approach to automatic region selection. In particular, we will see that knowledge about the underlying system can be efficiently exploited for finding good regions.

Concluding this chapter, we have seen the two core concepts of the abstraction part of our verification framework: predicate abstraction and spotlight abstraction, plus a number of valuable extensions. Our approach facilitates the construction of small abstract models that preserve several properties of the original system. The next question is – how to automatically arrive at the right level of abstraction for a certain verification task? – which we will approach in the subsequent chapter. Beforehand, we take a look at related work on abstraction in formal verification.

4.3 Related Work

Our research on abstraction for concurrent systems is connected to other approaches in a number of ways. In this section we summarise and extend our previous references to related works.

Abstraction is a key technique for reducing the complexity of formal verification which has received considerable attention in research. In early work [38], Clarke et al. transferred the concept of abstract interpretation [46, 47] to the field of temporal logic model checking. Their approach is based on the definition of an abstraction function mapping concrete data domains to abstract ones. This allows to construct a small abstract state model of a given

finite-state program. The obtained model conservatively approximates the set of reachable states, and thus, enables the verification of either existential or universal properties. A similar approach is Kurshans *localisation reduction* [91], also known as *variable hiding*. This reduction technique abstracts away entire variables with non-deterministic assignments. While the practical applicability of the abovementioned approaches is limited to the verification of smaller hardware programs, subsequent works focused on improvements of abstraction techniques towards software model checking.

Boolean predicate abstraction (compare Section 4.1.1) introduced by Graf and Saidi [66] and Ball et al. [11] is a method for automatically transforming guarded command systems [66] resp. C programs [11] into boolean programs. These boolean programs are defined over a finite set of predicates rather than over concrete variables, which leads to a significantly smaller state space. The transformation can either preserve existential or universal properties. *Three-valued predicate abstraction* (compare Section 4.1.2) proposed by Godefroid and Jagadeesan [65] is capable of preserving both existential *and* universal properties. This extension of classical boolean abstraction employs a third truth value *unknown* that facilitates to explicitly reveal the loss of information due to abstraction, and thus, to combine over- and underapproximation in one abstract model. Three-valued abstraction is used in many frameworks for formal verification, e.g. [115, 67, 4, 87], and in particular, it is one of the core concepts of the verification technique developed in this thesis. Besides, there exist more generalised approaches to predicate abstraction, based on four-valued [73] or generally multi-valued [99] domains for truth values. These extensions enable a more nuanced modelling of the loss of information, however, at the price of an increasing complexity of verification.

All previously considered approaches can be regarded as *data abstraction techniques* that cut down the state space induced by large variable domains. Concurrency is another major challenge in formal verification. Therefore, specific reduction techniques for concurrent systems have been investigated in a number of works. *Spotlight abstraction* was initially introduced by Wachter and Westphal [123] as an approach to the verification of parameterised systems. A spotlight is set on a small set of processes, while the remaining ones are summarised into one shade component, i.e. they are nearly completely abstracted away by the truth value *unknown*. This form of abstraction yields an overapproximation of the original system, and thus, does not preserve existential properties. Schrieb et al. [112] combined spotlight abstraction with three-valued predicate abstraction for verifying non-uniform concurrent systems (compare Section 4.2). Their approach guarantees the preservation of full CTL properties. Our framework for abstraction is based on the method from [112], however, we have enhanced the original approach in several ways. The new concepts *shade clustering* (compare Section 4.2.2) and *region summarisation* (compare Section 4.2.3) enable us preserve additional definite behaviour in the abstraction. Shade clustering can be regarded as a generalisation of the original spotlight principle. Instead of summarising the

shade processes into one component, we build multiple clusters of similar shade processes. Our region summarisation technique is related to the lazy abstraction approach by Henzinger et al. [79]. Like region summarisation, lazy abstraction involves different degrees of precision in different parts of the abstract model. However, we achieve this effect by summarising control flow locations, whereas lazy abstraction associates a different number of predicates with each control flow location.

There exist a number of reduction techniques for concurrent systems that follow a similar approach as spotlight abstraction. Counter abstraction by Pnueli et al. [110] considers one process in detail, whereas for the remaining system it is just counted whether *no*, *exactly one* or *more than one* process is at a particular control flow location. Environment abstraction proposed by Clarke et al. in [35] combines this approach with boolean predicate abstraction. However, these two techniques are restricted to parameterised systems consisting of identical processes. We address parameterised verification separately in Chapter 6.

A different approach to the abstraction of concurrent systems is compositional reasoning. In [78] Henzinger et al. present a verification framework based on *thread-modular abstraction*. Instead of constructing one abstract model for the entire concurrent system, each thread (i.e. process) is separately abstracted and verified under an overapproximating environment assumption. The single verification results are then used to infer a global property of the system. This technique is limited to the verification of safety properties.

So far, we have discussed abstraction techniques that are based on some form of approximation, i.e. not every result obtained for the abstract model can be transferred to the original system. Nevertheless, there also exist "precise" abstraction techniques for concurrent systems. Partial order reduction by Godefroid [64] is technique that exploits independence of concurrently executed operations. If the execution of a number of operations has the same result under each possible ordering then the corresponding state transition model is restricted to one representative ordering. This form of reduction preserves safety as well as liveness properties, while the number of possible interleavings is cut down. The effectiveness of partial order reduction correlates with the degree of independence in the concurrent system. In our abstraction framework, we also reduce the number of interleavings, however, not by exploiting independence but by summarising processes in approximative shade components. Hence, our approach to state space reduction is more rigorous but it involves a loss of precision. Another "precise" abstraction technique for concurrent systems is symmetry reduction [101] which is based on the exploitation of the inherent symmetries of parameterised systems. The combination of spotlight abstraction and symmetry reduction is the topic of Chapter 6.

Many of the abstraction techniques that we have discussed in this section are integrated into frameworks for abstraction refinement. Thus, we will consider some of them again in the next chapter, where we introduce our

approach to the refinement of imprecise abstractions. Moreover, we will discuss prospects of combining these related techniques with our approach in the section about future work (Section 8.3).

Chapter 5
Heuristic-Guided Abstraction Refinement

Abstraction is essential in handling the complexity of real-life concurrent software systems in temporal logic verification. In the previous chapter we have considered predicate abstraction and spotlight abstraction, and moreover, introduced a number of enhancements that enable us to build small abstract models which still preserve certain system properties we are interested in. However, in the case that abstraction-based model checking yields *unknown*, we cannot draw any conclusions about the concrete system. We solely know that the current abstraction is too coarse for a definite result in verification. Therefore, common approaches usually combine abstraction with iterative abstraction refinement. – Starting with a very coarse abstract model, new predicates (or processes) are gradually added until a level of abstraction is reached where the property of interest can be proven or refuted. Abstraction refinement can be performed manually which, however, contradicts the idea of a fully automatic approach to software verification. Automated refinement techniques are usually based on the elimination of spurious or unconfirmed counterexamples by adding further details to the abstraction. Typically, a counterexample hints at several ways of abstraction refinement, but not every possible refinement step is expedient. Unfavourable decisions may unnecessarily enlarge the abstract model, without leading to definite result in verification. Thus, automatic abstraction refinement is generally the crucial and most challenging part of abstraction-based verification.

In this chapter, we present the refinement procedure of our verification approach. We start with the introduction of *counterexample-guided abstraction refinement* (CEGAR), a well-established technique for the iterative refinement of abstract system models. Common CEGAR approaches are defined for boolean abstractions and base their refinement decisions on the analysis of *one* counterexample in each iteration. We show that for our three-valued abstractions *multiple* counterexamples can be efficiently generated, which gives us a broader basis for refinement decisions. Finally, we present our *heuristic* framework for abstraction refinement. Based on a structural analysis of the underlying system we heuristically evaluate the benefit of potential refinement

steps, which enables us to guide the refinement in expedient directions, and thus to obtain definite verification results on very small abstractions.

5.1 Counterexample-Guided Abstraction Refinement

Software verification based on boolean predicate abstraction commonly incorporates *counterexample-guided abstraction refinement* (CEGAR) [34]. New predicates are incrementally added to an initially very coarse abstract model until a level of abstraction is reached where the property of interest can be either proven or refuted. This fully automatic approach to iterative abstraction refinement is guided by *spurious counterexamples* – error paths that are only feasible in the abstract model, but not in the original system. In order to determine whether a counterexample that has been discovered in the abstraction is *real* or *spurious*, it has to be retraced on the concrete system. A spurious counterexample eventually reaches an abstract state that is *not* reachable in the original system. Now, classical counterexample-guided abstraction refinement automatically adds new predicates that separate this abstract state into more concrete states, and thus, eliminate the spurious counterexample. The new predicates are commonly derived from weakest preconditions [9, 13] or from Craig interpolants [95, 77, 96] computed for locations along the counterexample.

CEGAR frameworks are widespread in verification techniques based on boolean abstractions [34, 10, 18, 27]. The counterexample-guided abstraction refinement approach has also been transferred to the field of three-valued abstractions [112, 114, 67]. Here, the uncertainty in *unconfirmed counterexamples* is eliminated by adding new predicates [112, 114, 67] or also processes [112]. Three-valued abstraction refinement generalises the classical CEGAR. In this section, we therefore introduce the three-valued variant of counterexample-guided abstraction refinement. All the definitions and the general procedure presented here can be straightforwardly transferred to the boolean scenario. We start with a motivating example. In Figure 5.1 we have a message passing system where several processes communicate via channel c, but only $Proc_1$ attempts to communicate via channel d.

Here we might be interested in verifying whether $Proc_1$ never reaches the *error* location, i.e. whether the CTL formula $\mathbf{AG}\neg(pc_1 = 4)$ yields *true*. The error will be only reached if the communication operations $receive(c,x)$ and $receive(d,y)$ can be successfully executed, and if afterwards $x < y$ holds. However, there is no process that ever sends a message to channel d. Thus, receive(d,y) will be never successful and we can conclude that $\mathbf{AG}\neg(pc_1 = 4)$ holds. In the following, we show how this verification result can be systematically obtained by three-valued spotlight abstraction with counterexample-guided abstraction refinement. The first step is to select an initial abstraction, or rather, an *initial spotlight* (see Definition 5.1).

$$c,d : \textbf{channel}\ [1]\ \textbf{of integer}$$

$$Proc_1 :: \begin{bmatrix} \textbf{local}\ x : \textbf{integer where}\ x = 0 \\ \textbf{local}\ y : \textbf{integer where}\ y = 0 \\ 1: \quad \textbf{receive}(c,x) \\ 2: \quad \textbf{receive}(d,y) \\ 3: \quad \textbf{if}\ x < y\ \textbf{then} \\ \quad [\,4: \quad \textbf{error}\,] \\ \quad \textbf{else} \\ \quad [\,5: \quad \textbf{end}\,] \end{bmatrix}$$

$$\|_{i=2}^{n}\ Proc_i :: \begin{bmatrix} \dots \\ \textbf{send}(c,1) \\ \dots \end{bmatrix} \quad \|_{j=n+1}^{m}\ Proc_j :: \begin{bmatrix} \dots \\ \textbf{receive}(c,z_j) \\ \dots \end{bmatrix}$$

Fig. 5.1 Message passing system $Sys_{12} = \|_{i=1}^{n}\ Proc_i$ over $Var = \{c,d,x,y,z_{n+1},\dots,z_m\}$.

Definition 5.1 (Initial Spotlight).
Let $Sys = \|_{i=1}^{n}\ Proc_i$ be a concurrent system over a set of variables Var. Moreover, let ψ over Sys be the CTL formula to be verified. Then the *initial spotlight* $Spot = Spot(Proc) \cup Spot(Pred)$ is defined by the processes that are referenced in ψ, and the atomic predicates over Var that are subformulae of ψ.

Hence, for the message passing system Sys_{12} in Figure 5.1 and the CTL formula $\mathbf{AG}\neg(pc_1 = 4)$ we get $Spot(Proc) = \{Proc_1\}$ and $Spot(Pred) = \emptyset$ as the initial spotlight. Model checking the formula $\mathbf{AG}\neg(pc_1 = 4)$ on an abstract Kripke structure K^a corresponding to the initial spotlight yields *unknown*. Hence, it is uncertain whether this assertion about the branching structure of K^a holds or not. There exists no substructure or path π of K^a that *definitely* proves or refutes the formula. However, there must be a substructure with some *unknown* transitions or labellings that *may* correspond to a real counterexample or witness if we resolve some uncertainty by refinement. We call such a substructure an *unconfirmed counterexample*[1]. For convenience, we sometimes just write *counterexample* if it is clear from the context that we refer to an unconfirmed one. The unconfirmed counterexample π that we obtain for checking $\mathbf{AG}\neg(pc_1 = 4)$ on our current abstraction is depicted in Figure 5.2.

$$\pi :: (pc_1 = 1) \xrightarrow{\ \bot\ } (pc_1 = 2) \xrightarrow{\ \bot\ } (pc_1 = 3) \xrightarrow{\ \bot\ } (pc_1 = 4)$$

Fig. 5.2 Unconfirmed counterexample π for $\mathbf{AG}\neg(pc_1 = 4)$ on the abstraction of Sys_{12}, given by $Spot(Proc) = \{Proc_1\}$ and $Spot(Pred) = \emptyset$.

[1] Remember that by the term "unconfirmed counterexample", we refer to a substructure of a Kripke structure that can be extended to either a real counterexample or a real witness for a temporal logic formula. Moreover, remember that unconfirmed counterexamples can be always reduced to linear fragments, i.e. paths. (Compare Section 2.2)

As we can see, the abstract state space is solely defined over the program counter of the spotlight process $Proc_1$. The abstract computation represented by the counterexample starts at $Proc_1$'s initial location. It is *unknown* whether *receive* on channel c can be successfully accomplished, because the guard of this operation, $\neg empty_c$, is not part of the abstraction. The same holds for the subsequent *receive* on channel d. Finally, it is uncertain which branch of the *if-then-else* operation at location 3 can be taken, since the predicate $x < y$ is missing in the current abstraction. Thus, the *error* location will *maybe* be reached according to this counterexample.

Our informal analysis of the counterexample already gives us a rough idea about counterexample-guided abstraction refinement. The *unknown* transitions along the trace correspond to guarded operations in a computation, where the guards are currently *not* part of the abstraction. Adding one or more of these guards to the abstraction would remove some uncertainty and might even bring us a definite result in verification. Hence, $\neg empty_c$, $\neg empty_d$ and $x < y$ are the *refinement candidates* that we can derive here. For the sake of illustration, we henceforth will directly annotate our counterexamples with the candidates, i.e. for π we get:

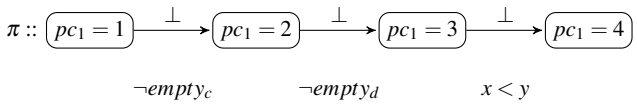

Fig. 5.3 Unconfirmed counterexample π for $\mathbf{AG}\neg(pc_1 = 4)$ on the abstraction of Sys_{12}, given by $Spot(Proc) = \{Proc_1\}$ and $Spot(Pred) = \emptyset$, annotated with refinement candidates.

Such annotated counterexamples can be automatically generated by our verification framework. In spotlight abstraction, candidates can not only be predicates but also processes. E.g., a predicate may be *unknown* in a state along a counterexample due to a transition associated with the shade component. Then each shade process modifying this predicate is a refinement candidate. From [112] we get the following procedure for determining refinement candidates:

Let $Sys = \|_{i=1}^{n} Proc_i$ be a concurrent system and $Spot = Spot(Proc) \cup Spot(Pred)$ a given spotlight. Let $K^a = (S^a, R^a, L^a, \mathbb{F}^a)$ be the corresponding three-valued Kripke structure over $AP = Spot(Pred) \cup \{pc_i = j \mid Proc_i \in Spot(Proc), j \in Loc_i\}$, and let $s^a \in S^a$ be a state of K^a. Moreover, let ψ be a CTL formula over AP with $[K^a, s^a \models \psi] =$ unknown and let π be a corresponding unconfirmed counterexample. Then the uncertainty in π may be caused by:

1. An *unknown* transition (s, s') along π. Let *bop* be the concrete basic operation associated with (s, s'). Then we can distinguish the following cases:

a. The guard e of bop is *not* contained in the set of predicates $Spot(Pred)$. Then e is a refinement candidate.

b. The guard e of bop is already contained in the set of predicates $Spot(Pred)$, but e is *unknown* in the state s. Then continue with step 2.

c. The transition (s,s') is associated with some shade component $Proc^{j}_{Shade}$. Then all processes summarised in $Proc^{j}_{Shade}$ are refinement candidates. (*Only with shade clustering. Compare Section 4.2.2.*)

2. A predicate $p \in Spot(Pred)$ that is *unknown* in some state s in π. Let $\bar{s}$ be the last predecessor of s in π, where p evaluates to a definite value. Moreover, let $\bar{s}'$ be the direct successor of $\bar{s}$ in π. Then we can distinguish the following cases:

   a. The transition $(\bar{s},\bar{s}')$ is associated with a spotlight process. Let bop be the basic operation corresponding to $(\bar{s},\bar{s}')$, let $wp_{bop}(p)$ be the weakest precondition of bop with regard to p, and $wp_{bop}(p)$ is *not* contained in the set of predicates $Spot(Pred)$. Then $wp_{bop}(p)$ is a refinement candidate.

   b. The transition $(\bar{s},\bar{s}')$ is associated with a spotlight process. Let bop be the basic operation corresponding to $(\bar{s},\bar{s}')$, let $wp_{bop}(p)$ be the weakest precondition of bop with regard to p, and $wp_{bop}(p)$ is already contained in the set of predicates $Spot(Pred)$, but $wp_{bop}(p)$ is *unknown* in the state $\bar{s}$. Then backtrack to step 2.

   c. The transition $(\bar{s},\bar{s}')$ is associated with some shade component $Proc^{j}_{Shade}$. Then all processes summarised in $Proc^{j}_{Shade}$ are refinement candidates. (*Note that in spotlight abstraction without shade clustering there is only one shade component.*)

Hence, refinement candidates are derived by computing weakest preconditions and by determining shade processes that are associated with certain transitions. Alternatively, candidates can be derived via Craig interpolation [95, 77, 96] which generates predicates that are relevant to eliminate the current counterexample. However, the latter approach is restricted to boolean abstractions where retracing a spurious error trace yields an unsatisfiable formula which is the basis for computing interpolants. In the following we thus assume that the refinement candidates are generated according to the procedure above.

It remains the question of how to automatically refine the abstraction, i.e. how to decide *which* candidate(s) should be drawn into the spotlight. A common way is to add *all* discovered candidates to the abstraction in each step, e.g. applied in [96]. Other CEGAR approaches, e.g. [34, 112, 114, 67], just determine the *first* refinement candidate along the counterexample and add this candidate to the abstract model. In both cases the steps *abstraction*,

model checking, and *refinement* are then iteratively repeated until a definite result in verification is obtained. We will discuss the advantages and disadvantages of the *all-candidates* and the *first-candidate* strategy in counterexample-guided abstraction later in detail. Beforehand, we continue with our running example Sys_{12} and refine the current abstraction $Spot(Proc) = \{Proc_1\}$ and $Spot(Pred) = \emptyset$ according to the *first-candidate* strategy.

The first candidate along the counterexample in Figure 5.3 is the guard $\neg empty_c$. Hence, we add the atomic predicate $empty_c$ to the abstraction. The resulting spotlight is $Spot(Proc) = \{Proc_1\}$ and $Spot(Pred) = \{empty_c\}$. Again, model checking yields an indefinite result and we obtain another annotated counterexample:

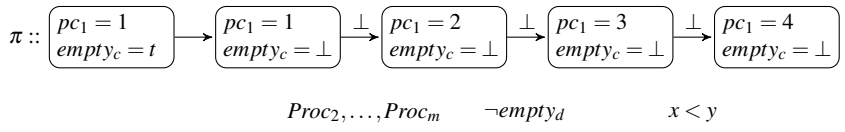

Fig. 5.4 Unconfirmed counterexample π for $\mathbf{AG}\neg(pc_1 = 4)$ on the abstraction of Sys_{12}, given by $Spot(Proc) = \{Proc_1\}$ and $Spot(Pred) = \{empty_c\}$, annotated with refinement candidates.

The first transition of this trace is associated with the shade component. It sets the initially *true* predicate $empty_c$ to $\bot$, which characterises that there are processes in the shade that potentially communicate via the channel c, and hence, may affect $empty_c$ in an *unknown* manner. In the consequent state it is thus *uncertain* whether *receive* on channel c can be successfully accomplished by $Proc_1$. Our counterexample reveals that every shade process communicating on c is a corresponding refinement candidate. The remaining candidates $\neg empty_d$ and $x < y$ are the same as in our previous counterexample.

Continuing the *first-candidate* strategy for counterexample-guided abstraction refinement, we iteratively add the processes affecting $empty_c$ to the spotlight until we have validated or refuted that there exists a transition from $Proc_1$'s location 1 to 2.[2] In case that abstraction refinement reveals such a computation where $send(c,x)$ is definitely successfully executed by $Proc_1$, we will obtain a further counterexample, now with $\neg empty_d$ as the first candidate. – The refinement candidate $empty_d$ is the crucial predicate here: There is no process in the message passing system that ever sends a message via channel d. Hence, adding $empty_d$ to the spotlight (together with applying region summarisation[3] for the locations 1 and 2) will immediately reveal that the error location is unreachable, and we can deduce that $\mathbf{AG}\neg(pc_1 = 4)$ holds.

[2] Note that, due to simplification, the processes $Proc_2$ to $Proc_m$ are not entirely specified in our example system Sys_{12}, and thus, we can not predict how many of these processes have to be added to the spotlight in order to validate or refute a transition from location 1 to 2 in $Proc_1$.

[3] In Section 5.2 we will see how the selection of regions can be efficiently automated.

We can summarise spotlight abstraction with iterative counterexample-guided abstraction refinement as follows:

Let $Sys = \|_{i=1}^{n} Proc_i$ *be a concurrent system and* ψ *the CTL formula to be verified. Then:*

1. Determine the initial spotlight $Spot = Spot(Proc) \cup Spot(Pred)$ according to Definition 5.1.

2. While no definite result in verification is obtained:

   a. Build the abstract Kripke structure K^a corresponding to Sys and $Spot$.

   b. Let s^a be the state of K^a that corresponds to the initial configuration of Sys. Check $[K^a, s^a \models \psi]$.

   c. If model checking yields a definite answer, this result can be transferred to the original system Sys,

   d. else, model checking returns an unconfirmed counterexample π. Select a refinement candidate along π according to some strategy and add the candidate to the spotlight.

Hence, with counterexample-guided abstraction refinement we have a systematic and fully automatable approach for incrementally constructing abstractions in temporal logic model checking. However, our running example reveals that following the widely spread *first-candidate* strategy may yield abstractions that are unnecessarily large: None of the considered processes $Proc_2$ to $Proc_m$ are essential for validating $\mathbf{AG}\neg(pc_1 = 4)$, whereas the lastly added predicate $empty_d$ by itself suffices for a definite result in verification.

Nevertheless, there exist arguments that justify the *first-candidate* approach under certain circumstances. Remember that classical boolean abstraction techniques require a retracement of counterexamples on the original system, in order to decide whether a counterexample is real or spurios. This additional computation involves a partial exploration of the concrete state space, and thus, suffers from state explosion. However, the retracement can be immediately stopped when the counterexample reaches a state that is not reachable in the original system. Thus, the *first-candidate* strategy in boolean CEGAR can significantly reduce the effort for retracing counterexamples. Moreover, it has been demonstrated by Clarke et al. [34] that selecting the first candidate is an appropriate heuristic for obtaining the coarsest refinement that eliminates a spurious counterexample. These arguments apply to boolean abstractions. However, in three-valued abstractions we have slightly different circumstances. First, there is *no* necessity for retracing counterexamples. – A retracement of counterexamples in the context of spotlight abstraction rather unreasonable, since abstract transitions associated with the shade cannot be mapped to a

single concrete operation. And second, refinement for three-valued abstractions is *not* limited to the elimination of counterexamples – an unconfirmed counterexample might also be *confirmed* by refining the abstraction. Thus, a refinement heuristic which is exclusively geared towards the cheapest way to eliminate a counterexample appears to be too narrowed for the three-valued scenario.

CEGAR frameworks based on interpolation, e.g. [77, 96], usually follow a more elaborate approach to refinement. These techniques generate a so-called Craig interpolant for each control location along a spurious error trace. Such an interpolant corresponds to a predicate (expression) that is relevant at the particular location in terms of proving the infeasibility of the trace. All interpolants computed for a counterexample are then *locally* added to the abstraction, i.e. only at abstract states associated with the control location where the respective interpolant was generated. This yields an abstraction with different degrees of precision in different states, which is commonly known as *lazy abstraction* [79, 98]. This approach to refinement can be regarded as an improved *all-candidate* strategy that involves a local enlargement of the abstract state space – just along the spurious error trace. Such local refinements have been proven to be highly efficient for counterexample elimination. However, in our work we follow a broader approach. As already mentioned, our unconfirmed counterexamples generally reveal *two* directions for abstraction refinement, either towards *confirmation* or towards *refutation*. Moreover, we are focused on finding refinements that eventually lead us to a definite result in verification – which may not necessarily correlate with the elimination of a single counterexample. There are also some technical aspects that prevent the straightforward applicability of interpolation-based refinement to our scenario: Interpolation-based abstraction refinement is limited to the analysis of *finite* error traces, i.e. to the verification of safety properties – whereas we focus on safety *and* liveness properties. Furthermore, Craig interpolation requires to retrace a counterexample on the original system in order to generate an unsatisfiable boolean formula that can be decomposed into interpolants. However, our approach profits from a retracement-free refinement procedure, and our three-valued encoding does not reveal a boolean (un)satisfiability problem.

Existing approaches to three-valued abstraction refinement, e.g. [112, 114, 67], have generalised the native concept of boolean counterexample-guided abstraction refinement [34]. These techniques have been proven to be sound, and their practical applicability has been demonstrated. However, hardly any research has been spent on optimisations and adaptations. Decision procedures for refinements of three-valued abstractions are directly adopted from the basic CEGAR approach, and thus, do not exploit any specific characteristics of the three-valued scenario.

In this work, we have developed a heuristic-guided abstraction refinement framework that is specifically tailored to three-valued abstractions of concurrent systems. Our refinement heuristics also incorporate counterexamples,

but *additionally* the dependency structure of the considered system, and the uncertainty in the abstraction. Moreover, our heuristics are not geared towards a local goal like eliminating a certain counterexample, but towards finally obtaining the coarsest abstraction that is precise enough for a definite result in verification. In the remainder of this chapter, we gradually introduce our heuristic approach to three-valued abstraction refinement for concurrent systems. In the first step, we show that it can be beneficial to base refinement decisions on *more than one* counterexample.

5.2 Multiple Counterexample-Generation

Our approach to the verification of concurrent system aims at an abstraction refinement procedure that heuristically selects the presumably 'best' refinement candidate in each iteration. The selectable candidates are derived from a generated counterexample. Such a counterexample may hint at a large number of predicates and processes that are potentially relevant for the underlying verification task. Nevertheless, it is also possible that a counterexample only reveals a single candidate, and thus, there remains no scope for heuristic decisions. Common model checking tools, e.g. HSF-SPIN [56], are tailored to find *short* counterexamples, which is motivated by the fact that short traces are easier to interpret by a user, and furthermore, require less exploration of the state space. However, in our scenario the 'interpretation', i.e. counterexample-guided abstraction refinement, is done automatically – not by a user. Moreover, we aim at finding *small and precise abstractions* by selecting good refinement candidates – and not primarily at a small number of explored abstract states. Short counterexamples in particular carry the risk of a too narrowed number of associated candidates. Hence, if there exist more than one (short) counterexample for the checked property in the current abstraction, then it can be advisable to consider the additional longer traces as well for deriving refinement candidates. In this section, we introduce *multiple counterexample-generation* for heuristic-guided abstraction refinement, which we have implemented on top of the model checker χChek [30, 54] that was originally geared towards finding short counterexamples. We start with a motivating example.

We look again at the message passing system Sys_{12} in Figure 5.1 and check whether $Proc_1$ will eventually reach *end*, i.e. whether $\mathbf{AF}(pc_1 = 5)$ holds. For the initial spotlight $Spot(Proc) = \{Proc_1\}$ and $Spot(Pred) = \emptyset$, model checking yields *unknown*, and moreover returns the counterexample π in Figure 5.5. According to the trace π, $Proc_1$ will *maybe* remain at location 1 forever. As we can see, π solely hints at the candidate $empty_c$. Thus, our next refinement step is predetermined. We add the predicate $empty_c$ to the spotlight. And again, we obtain a an indefinite result in verification and another counterexample π' in Figure 5.6.

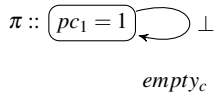

Fig. 5.5 Unconfirmed counterexample π for $\mathbf{AF}(pc_1 = 5)$ on the abstraction of Sys_{12}, given by $Spot(Proc) = \{Proc_1\}$ and $Spot(Pred) = \emptyset$, annotated with the only refinement candidate.

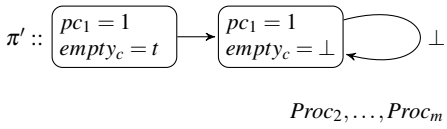

Fig. 5.6 Unconfirmed counterexample π' for $\mathbf{AF}(pc_1 = 5)$ on the abstraction of Sys_{12}, given by $Spot(Proc) = \{Proc_1\}$ and $Spot(Pred) = \{empty_c\}$, annotated with refinement candidates.

The path π' characterises a computation where the shade component sets $empty_c$ to *unknown*, and thus, $Proc_1$ will *maybe* remain at location 1 permanently. Furthermore, π' reveals a number of shade processes as refinement candidates. We can choose between the processes $Proc_2$ to $Proc_m$ for the next refinement step. However, we might end up in iteratively adding all these processes to the spotlight until we discover a counterexample that hints at the candidate $empty_d$ – which is again the crucial predicate for a definite result in the current verification task. We see that not only unfavourable refinement decisions, but also unfavourable counterexamples are an issue in iterative abstraction refinement. Even the best heuristic would be worth nothing if the generated counterexample reveals only dispensable refinement candidates. A prerequisite for the successful application of heuristics in abstraction refinement is thus a wide range of candidates. In our approach, we therefore generate *multiple* counterexamples in each iteration, and then use *all* of them for deriving refinement candidates. For the initial spotlight $Spot(Proc) = \{Proc_1\}$, $Spot(Pred) = \emptyset$ and the temporal logic formula $\mathbf{AF}(pc_1 = 5)$ we can identify three distinct counterexamples (see Figure 5.7).

The first, π^1, is the same as in Figure 5.5; the second, π^2, characterises that $Proc_1$ *maybe* successfully receives on channel c, and then *maybe* remains at location 2 forever, due to an empty channel d. And the last, π^3, corresponds to a *potential* computation where $Proc_1$ terminates at the *error* location. These counterexamples together give us an enlarged set of refinement candidates, which in particular contains the crucial candidate $empty_d$. In the next section, we will show how we determine the presumably 'best' refinement candidate by heuristic evaluation. Beforehand, we take a deeper look at the *systematic* generation of multiple counterexamples in our abstraction refinement framework.

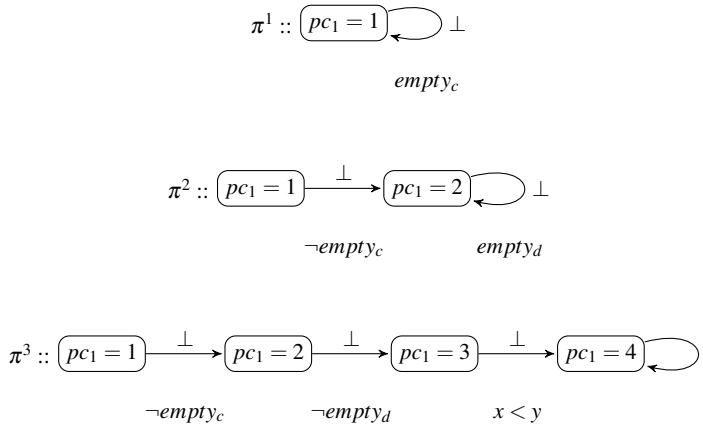

Fig. 5.7 Unconfirmed counterexamples π^1, π^2 and π^3 for $\mathbf{AF}(pc_1 = 5)$ on the abstraction of Sys_{12}, given by $Spot(Proc) = \{Proc_1\}$ and $Spot(Pred) = \emptyset$, annotated with refinement candidates.

In Figure 5.8 we have a tree representation T of the abstract state space corresponding to our running example with $Spot(Proc) = \{Proc_1\}$ and $Spot(Pred) = \emptyset$. The root node characterises the initial state of the abstract system. In the leaf nodes, a cycle in the state space has been reached. As we can see, all counterexamples that we have considered before are comprised in T.

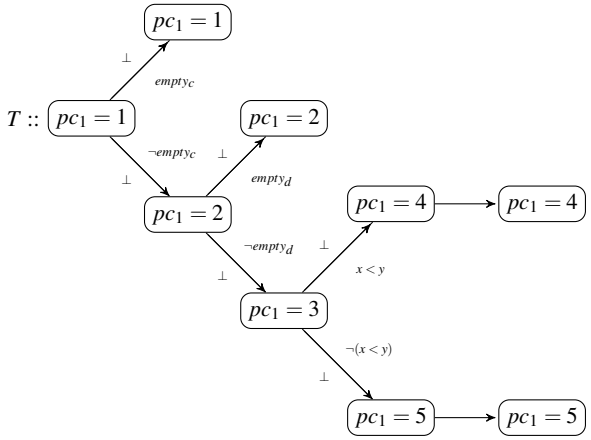

Fig. 5.8 State space exploration tree T for the abstraction of Sys_{12}, given by $Spot(Proc) = \{Proc_1\}$ and $Spot(Pred) = \emptyset$.

In case that model checking yields *unknown*, such an abstract state space tree is partially constructed and explored in order to generate an unconfirmed

counterexample. Our employed model checker χChek is tailored to find short counterexamples. Hence, for our running example the path π^1 will be detected first (see Figure 5.9).

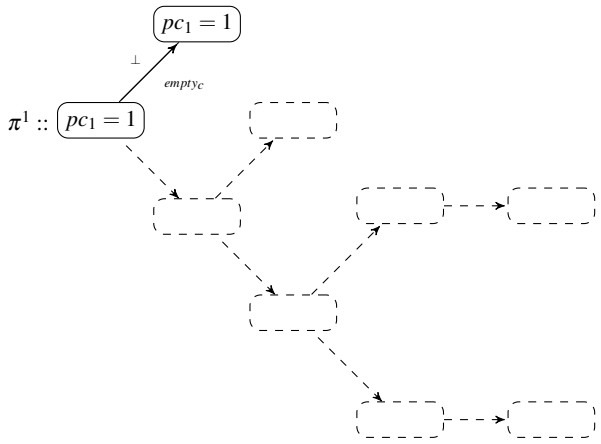

Fig. 5.9 Counterexample π^1 in the state space exploration tree T.

However, we have already discussed that π^1 is inexpedient for counterexample-guided abstraction refinement. In our approach to *multiple* counterexample-generation we thus backtrack to so-called *open branching states* of already discovered counterexamples in order to start a search for further counterexamples in the same abstraction:

Let $Sys = \|_{i=1}^{n} Proc_i$ be a concurrent system, ψ the CTL formula to be verified, and $Spot = Spot(Proc) \cup Spot(Pred)$ a given spotlight where model checking ψ yields unknown. *Moreover, let T be a tree representation of the corresponding abstract state space. Then, starting at the root node of T, the abstract state space is gradually explored until an unconfirmed counterexample π has been discovered. For each $\perp$-transition (π_i, π_{i+1}) along π that is associated with a branch (e.g. an* if *branch with guard e) of a spotlight process $Proc_j$ the corresponding complementary branch (π_i, π_k) (e.g. an* else *branch with guard $\neg e$) is determined. The state π_i is then marked as an* open branching state *of π with the alternative branch (π_i, π_k). For multiple counterexample-generation the procedure backtracks to such an open branching state π_i, marks it as* closed*, and attempts to expand the prefix $\pi_0 \ldots \pi_i \pi_k$ to another unconfirmed counterexample. This step is then iteratively repeated until no further unconfirmed counterexample can be detected or a bound for the number of counterexamples has been reached.*

Hence, for our running example we backtrack to the initial state of our first counterexample π^1 and then take the alternative branch to the abstract state $pc_1 = 2$. Now, χChek returns the trace π^2 (see Figure 5.10).

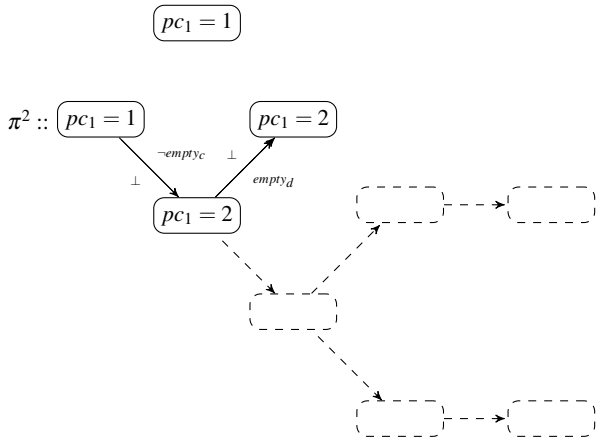

Fig. 5.10 Counterexample π^2 in the state space exploration tree T.

The counterexample π^2 gives us the additional refinement candidate $empty_d$. For the generation of further counterexamples, we backtrack to the state $pc_1 = 2$ and take the previously undiscovered branch to $pc_1 = 3$. We obtain the counterexample π_3 (see Figure 5.11) which hints at another new candidate $x < y$.

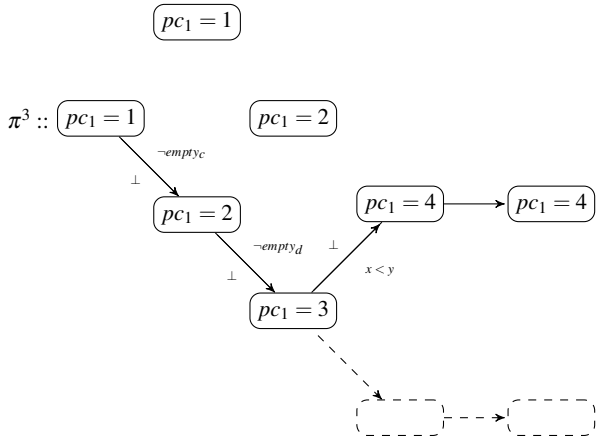

Fig. 5.11 Counterexample π^3 in the state space exploration tree T.

The remaining branch from $pc_1 = 3$ to $pc_1 = 5$ can not be extended to an error trace. Thus, there exist no further unconfirmed counterexample in the current abstraction. Nevertheless, with π^1, π^2 and π^3 we have a sufficiently broad basis for counterexample-guided abstraction refinement. In general, our approach to multiple counterexample-generation yields error traces that

cover *different* branches of the control flow of the spotlight processes. This involves a large variety of the corresponding refinement candidates. Moreover, it enables us to identify candidates that can be used for the concretisation of *more than one* unconfirmed counterexample.

Multiple counterexample-generation also facilitates the automatic selection of expedient regions (compare Section 4.2.3). Obtaining a second counterexample is always the result of backtracking to an open branching state and taking the alternative branch. In particular, this approach yields *complementary counterexamples*.

Definition 5.2 (Complementary Counterexamples and Associated Region). Let $Sys = \|_{i=1}^{n} Proc_i$ be a concurrent system, ψ the CTL formula to be verified, and $Spot = Spot(Proc) \cup Spot(Pred)$ a given spotlight where model checking ψ yields *unknown*. Moreover, let K be the corresponding three-valued Kripke structure. Let $\pi = \pi_0\pi_1\pi_2\ldots$ and $\pi' = \pi'_0\pi'_1\pi'_2\ldots$ be two different unconfirmed counterexamples in K for ψ. Then π and π' are *complementary* iff

- $\exists i \in \mathbb{N}$ such that $\forall 0 \leq k \leq i : \pi_k = \pi'_k$, i.e. both paths have a common prefix,
- $R(\pi_i, \pi_{i+1}) = \bot$ and $R(\pi_i, \pi_{i+1})$ corresponds to an operation op of a spotlight process $Proc_j$ with a guard e,
- $R(\pi'_i, \pi'_{i+1}) = \bot$ and $R(\pi'_i, \pi'_{i+1})$ corresponds to an operation op' of a spotlight process $Proc_j$ with a guard $\neg e$.

The *associated region Reg* of such a pair of complementary counterexamples π and π' is the set of control locations of $Proc_j$ that are reached along the suffixes $\pi_i\pi_{i+1}\pi_{i+2}\ldots$ and $\pi'_i\pi'_{i+1}\pi'_{i+2}\ldots$.

Hence, the counterexamples π^1 and π^2 in our running example are complementary. Both have the same initial state, one takes the branch to location 1 of $Proc_1$, whereas the other takes the complementary branch to location 2 of the same process. The predicate $empty_c$ is not part of the current abstraction, thus, both branching transitions evaluate to *unknown*. It is not predictable *which* of the two transitions can be actually taken. However, since both have complementary guards, either location 1 or location 2 will be reached. In case that we discover such a pair of *complementary counterexamples*, the sum of locations on both branches hints at an appropriate region. We e.g. have that π^1 and π^2 are complementary with regard to a branch of the process $Proc_1$. Thus, by summing up $Proc_1$'s locations on both branches, we obtain the region $Reg = \{1,2\}$. Applying region summarisation (compare Section 4.2.3) for Reg gives us a modified abstraction which is now completely independent from the channel c. This enables us to validate $\mathbf{AF}(pc_1 = 5)$ by just adding the predicate $empty_d$ to the spotlight. The general approach to spotlight abstraction with *multiple* counterexample-guided abstraction refinement and *automatic region summarisation* now works as follows:

Let $Sys = \|_{i=1}^{n} Proc_i$ be a concurrent system and ψ the CTL formula to be verified. Then:

1. Determine the initial spotlight $Spot = Spot(Proc) \cup Spot(Pred)$.

2. While no definite result in verification is obtained:

   a. Build the abstract Kripke structure K^a corresponding to *Sys* and *Spot*.

   b. Let s^a be the state of K^a that corresponds to the initial configuration of *Sys*. Check $[K^a, s^a \models \psi]$.

   c. If model checking yields a definite answer, this result can be transferred to the original system *Sys*,

   d. else, model checking returns a set of unconfirmed counterexamples $\Pi = \{\pi^1, \ldots, \pi^k\}$.

      i. Select a pair of complementary counterexamples out of Π, determine an associated region *Reg* and apply region summarisation.

      ii. If model checking yields a definite answer for the modified abstraction, this result can be transferred to the original system *Sys*,

      iii. else, remove the region.

   e. Select a refinement candidate along a counterexample in Π and add the candidate to the spotlight.

Thus, the final – and most crucial – step in each iteration of our enhanced CEGAR approach is the selection of a refinement candidate. In the next, we introduce our framework for *heuristic-guided* abstraction refinement. In particular, we show how the candidates derived by multiple counterexample generation can be heuristically evaluated in order to determine the presumably 'best' candidate for refinement.

5.3 Heuristic Framework for Abstraction Refinement

Multiple counterexample generation commonly yields a large set of refinement candidates in each abstraction iteration. Nevertheless, it remains to decide which *particular* candidate to select for refinement. We have already discussed that a naive approach like the *first-candidate* strategy may guide the refinement in unfavourable directions, without converging to a definite result in verification. Likewise, adding *all* derived candidates in one refinement step

may introduce a considerable amount of redundancy in the abstraction or may even cause the model checker to run out of memory. The unnecessary blow-up of the state space can be kept within reasonable bounds by applying lazy abstraction, i.e. by adding the candidates locally along the counterexample trace to the abstraction. However, such a local refinement is only feasible for predicates. But counterexamples may also hint at entire processes – which can only be added globally to the abstraction. Moreover, the idea of lazy abstraction contradicts our paradigm of an anticipating refinement. While lazy abstraction refinement focuses on the complete elimination of a single counterexample by locally adding multiple candidates in each step, we aim at gradually selecting single candidates that are globally valuable for our abstraction (e.g. for the concretisation of multiple counterexample traces), and thus, guide us closer to a definite result in the overall verification task.

In this section we introduce our framework for refining abstractions of concurrent systems. The capability of an abstraction refinement-based verification technique crucially depends on the quality of its decision procedure for selecting refinement candidates. Here we show how the selection of the 'most promising' candidate can be enhanced by heuristic guidance. Applying heuristics generally means utilising easily accessible information about the underlying problem in order to find good solutions. In our scenario the *problem* is the considered verification task, whereas a *good solution* corresponds to a minimal abstraction that is precise enough for a definite outcome in verification. Such an abstraction is, again, the result of a sequence of good refinement decisions. The *accessible information* that we exploit for our heuristic decisions particularly concerns structural aspects of the considered system, or more precisely, *dependencies* within the system and its current abstraction.

5.3.1 Abstraction Dependence Analysis

Communication is an inherent aspect of concurrent systems. Processes read variables modified by other processes, or receive messages sent by communication partners. Thus, the behaviour of a single process generally *depends* on the behaviour of the other processes in the system. Our heuristic framework for abstraction refinement is based on an iterative analysis of these dependencies within the considered system. The analysis in particular incorporates the current abstraction (i.e. the current separation of the system's components into spotlight and shade) and the generated set of refinement candidates. We will show that our dependence analysis can be efficiently performed, and the obtained results can be effectively exploited for the heuristic evaluation of refinement candidates.

Again, we illustrate our approach with the running example Sys_{12} (compare Figure 5.1). For the spotlight abstraction given by $Spot(Proc) = \{Proc_1\}$ and $Spot(Pred) = \emptyset$, model checking $\mathbf{AF}(pc_1 = 5)$ yields *unknown*. We generate

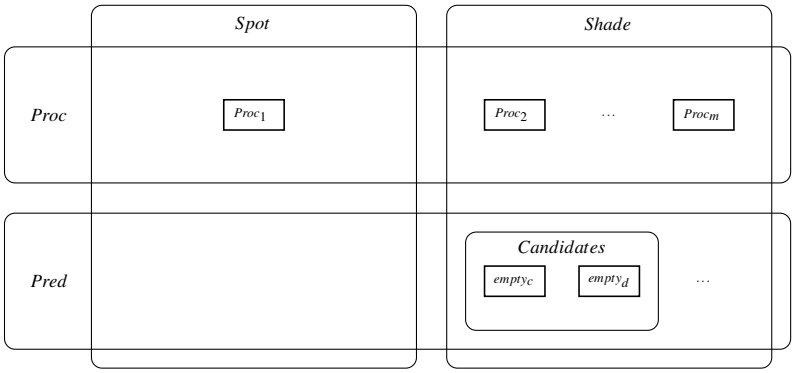

Fig. 5.12 Spotlight abstraction of Sys_{12} given by a partition of the system's processes and predicates into *Spot* and *Shade*. The refinement candidates $empty_c$ and $empty_d$ derived from the counterexamples π^1 and π^2 (compare Figures 5.9 and 5.10) correspond to a finite subset of the shade.

two counterexamples π^1 and π^2 (compare Figures 5.9 and 5.10), which hint at the set of refinement candidates $Candidates = \{empty_c, empty_d\}$. Figure 5.12 visualises our current abstraction, and moreover emphasises that the candidates are a finite subset of the shade. Such a spotlight abstraction together with a corresponding set of refinement candidates is the basis of our dependence analysis. For actually performing the analysis, we first require a formal notion of dependencies in abstractions of concurrent systems. Such dependencies arise from the *definition* and *reference* of variables.

Definition 5.3 (Definition and Reference of Variables).
Let $Sys = \|_{i=1}^{n} Proc_i$ be a concurrent system over a set of variables Var, and let $Pred$ be the set of all predicates over Var. We introduce the following sets for each process $Proc_i$ in Sys:

- $DEF(Proc_i)$ is the set of variables/channels that are defined (modified) by an operation of $Proc_i$,
- $REF(Proc_i)$ is the set of variables/channels that are referenced by an operation of $Proc_i$.

Furthermore, we introduce the following set for each predicate $p \in Pred$:

- $REF(p)$ is set of variables/channels that are referenced by p.

These sets enable us to formally characterise dependencies in spotlight abstractions of concurrent systems. For this, we use a graph-based representation – an *abstraction dependence graph* (ADG).

Definition 5.4 (Abstraction Dependence Graph).
An *abstraction dependence graph* is a tuple $ADG = (V, D)$ where

- V is a finite set of vertices,
- $D \subseteq V \times V$ is a set of directed edges, called *dependence relation*.

For a given spotlight abstraction and a set of refinement candidates we can construct the corresponding abstraction dependence graph according to the following definition.

Definition 5.5 (Spotlight Abstractions as Abstraction Dependence Graphs).
Let $Sys = \|_{i=1}^{n} Proc_i$ be a concurrent system with a spotlight abstraction given by the sets $Spot = Spot(Proc) \cup Spot(Pred)$ and $Shade = Shade(Proc) \cup Shade(Pred)$. Moreover, let ψ be a CTL formula that evaluates to *unknown* on this abstraction, let Π be an associated set of unconfirmed counterexamples, and let $Candidates \subseteq Shade$ be a set of refinement candidates derived from Π. The corresponding abstraction dependence graph is a tuple $ADG = (V, D)$ with

- $V := Spot \cup \underbrace{Candidates \cup Shade(Proc)}_{\subseteq Shade},$
- $D :=$
  $\{(v, v') \mid (v \neq v') \wedge$
  $((v \in Proc \wedge \exists\, x (x \in DEF(v) \wedge x \in REF(v')))$
  $\vee \ (v \in Pred \wedge v' \in Proc \wedge \exists\, x (x \in REF(v) \wedge x \in REF(v'))))\}.$

Thus, the vertices of an abstraction dependence graph correspond to the abstraction's components (processes as well as predicates) and the edges represent dependencies between the components. We restrict the set of vertices to the components that are relevant for our heuristic framework: the entire spotlight and a finite subset of the shade consisting of the current refinement candidates and all shade processes. Hence, constructing an ADG always yields a finite graph. The abstraction dependence graph corresponding to our running example is depicted in Figure 5.13.

As we can see, the ADG extends our visualisation of the spotlight abstraction (see Figure 5.12) with a dependence relation. If the relation $D(v, v')$ holds for a pair of vertices $v, v' \in V$ then we say: v' *depends* on v, or conversely: v *affects* v'. In our example, we e.g. have that the process $Proc_1$ depends on the predicate $empty_c$, or rather, $empty_c$ affects $Proc_1$. Dependencies of particular interest are those between individual candidates and the spotlight. A candidate that affects a large number of spotlight components is likely a *beneficial* choice for refinement. Adding it to $Spot$ would enrich the abstraction with new details that are relevant for *many* spotlight processes and predicates – and not only for the concretisation of a single counterexample. Thus, this choice might guide us closer to a definite result in verification. Contrary, a

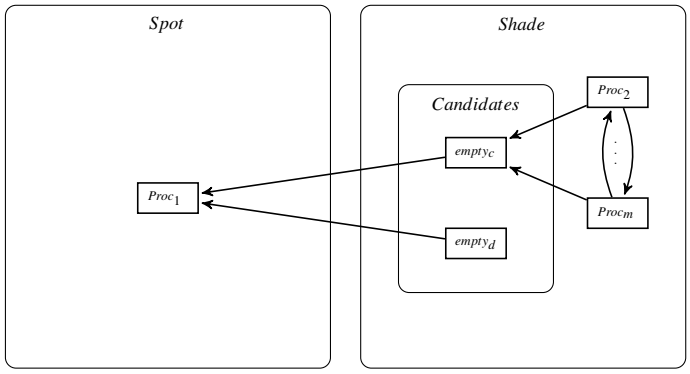

Fig. 5.13 Abstraction dependence graph corresponding to the spotlight abstraction given by $Spot(Proc) = \{Proc_1\}$ and $Spot(Pred) = \emptyset$ of the concurrent system Sys_{12}, and the set of refinement candidates $Candidates = \{empty_c, empty_d\}$. The depicted edges are restricted to the dependencies that are relevant for our heuristic framework: dependencies between the candidates and the spotlight, and dependencies within the shade.

candidate with lots of dependencies within the shade might be a *costly* choice. By selecting such a candidate for refinement we would introduce several new dependencies between the spotlight and the shade, i.e. several new *unknowns* in the abstraction.

In message passing systems we can distinguish different kinds of dependencies: Processes that communicate on the same channel generally depend on each other. More precisely, pairs of processes such that one process is a *sender* and the other is a *receiver* on a certain channel are potential *partners*, whereas two processes that both send (or both receive) on the same channel are *competitors*. Both, *partner* as well as *competitor* relationships between processes are bidirectional. However, for a pair of partners, one is the *sending partner* and the other is the *receiving parter*. The following definition formalises these different kinds of dependencies in message passing systems:

Definition 5.6 (Dependencies in Message Passing Systems).
Let $ADG = (V, D)$ be an abstraction dependence graph corresponding to a message passing system $Sys = \|_{i=1}^{n} Proc_i$. Moreover, let $Chan$ be the set of communication channels of Sys, and let $Proc \subseteq V$ be the subset of vertices that represent processes of Sys. We introduce the following relations on $Proc$:

- $Comp :=$
  $\{(v, v') \mid v, v' \in Proc \ \wedge \ D(v, v')$
  $\wedge \exists c \in Chan : (v \text{ and } v' \text{ send on } c) \text{ or } (v \text{ and } v' \text{ receive on } c)\}$

- $SendingPartner :=$
  $\{(v,v') \mid v,v' \in Proc \wedge D(v,v')$
  $\wedge \exists c \in Chan : v$ sends on c and v' receives on $c\}$

- $ReceivingPartner :=$
  $\{(v,v') \mid v,v' \in Proc \wedge D(v,v')$
  $\wedge \exists c \in Chan : v$ receives on c and v' sends on $c\}$

- $Partner := SendingPartner \cup ReceivingPartner$

For our running example we e.g. have that the sending shade processes $Proc_2$ to $Proc_n$ are competitors among themselves, the receiving shade processes $Proc_{n+1}$ to $Proc_m$ are also competitors among themselves, whereas all dependencies between these two groups correspond to partner relations. Of course, due to multiple channels and multiple communication statements per process, partners might be in turn competitors and a sending partner of a certain process might be in turn its receiving partner. However, we do not have these cases in our running example. Distinguishing partner and competitor dependencies is particularly useful in the analysis of *parameterised* message passing systems, which will be discussed separately in Section 6.5. In the remainder of this chapter we focus on the standard notion of dependency.

Before we take a detailed look at our refinement heuristics in the subsequent section, we want to consider a first simple example for the heuristic evaluation of candidates. As we just stated, *costs* and *benefits* of potential refinement steps arise from dependencies, or rather, from *incoming* and *outgoing* edges of each candidate in the abstraction dependence graph:

Definition 5.7 (Incoming and Outgoing Edges).
Let $ADG = (V,D)$ be an abstraction dependence graph, and V' be a subset of V. We introduce the following sets for each vertex $v \in V$:

- $IN(v,V')$ is the set of incoming edges into v from vertices in V',

- $OUT(v,V')$ is the set of outgoing edges from v into vertices in V'.

These sets let us define a very simple evaluation function $h : Candidates \to \mathbb{N}$ where

$$h(v) := \underbrace{|OUT(v,Spot)|}_{benefit(v)} - \underbrace{|IN(v,Shade)|}_{cost(v)}$$

i.e. for a candidate v we compute the number of outgoing edges into the spotlight and subtract the number of incoming edges from the shade. The refinement procedure selects the candidate with the best evaluation value: $arg\ max_{v \in Candidates}\ h(v)$. Thus, in our example $empty_d$ is chosen due to fewer (no) dependencies within the shade. The subsequent verification run on the

refined abstraction already reveals a definite result and refutes the property $\mathbf{AF}(pc_1 = 5)$ (compare Section 5.2).

This first example illustrates that even with heuristics based on a very simple dependence analysis we can guide the refinement in expedient directions, and thus obtain definite verification results on very small abstractions. Nevertheless, our overall approach aims at the verification of systems with a higher complexity than that of our small running example Sys_{12}. Hence, in our actual refinement framework we follow an enhanced approach to heuristic guidance. In particular, we construct abstraction dependence graphs extended with *weighted edges and vertices*.

Definition 5.8 (Weighted Abstraction Dependence Graph).
A *weighted abstraction dependence graph* is a tuple $WADG = (V, W, D)$ where

- V is a finite set of vertices,
- $W = \{W_1, \ldots, W_n\}$ is a set of weight functions for vertices with $\forall\, W_i \in W$: $W_i : V \to \mathbb{R}$,
- $D_w : V \times V \to \mathbb{N}$ is a *weighted dependence function*.

Edge weights allow us to comprise *quantitative aspects* in the dependence analysis. In our framework we hereby quantify the number of variables/channels that are shared between the abstraction's components, i.e. our dependence function D_w is defined as follows:

$$D_w(v, v') := \begin{cases} |DEF(v) \cap REF(v')| & \text{if } v \neq v' \text{ and } v \in Proc \\ |REF(v) \cap REF(v')| & \text{if } v \neq v' \text{ and } v \in Pred \\ 0 & \text{else} \end{cases}$$

By vertex weights we can also express benefits and costs *apart from dependence*: A beneficial aspect of a refinement candidate is, e.g., the number of occurrences as a candidate in the generated set of counterexamples, whereas the size of a candidate (for processes: the number of its control flow locations) is a cost factor. For a given spotlight abstraction the weight functions of the corresponding WADG can be implemented in several ways – depending on *which* aspects of the system should be particularly accentuated in the heuristic evaluation. In the following section we introduce our framework for the heuristic evaluation of refinement candidates in detail. In particular, we will see how heuristic refinement decisions can be improved by the incorporation of quantitative characteristics of the underlying system.

5.3.2 Heuristic Evaluation of Refinement Candidates

In the previous sections we have introduced the prerequisites for establishing a heuristic-guided abstraction refinement framework for the verification of concurrent systems: The *counterexample-guided abstraction refinement* approach provides us with a general frame for iteratively refining three-valued abstractions. With *multiple counterexample-generation* we have a technique for producing large sets of refinement candidates. Moreover, *abstraction dependence graphs* enable us to analyse and quantify several characteristics of concurrent systems and their abstractions. Finally, we want to see how these characteristics can be effectively incorporated into the heuristic evaluation of refinement candidates.

We start with a brief recap of the first steps of heuristic-guided abstraction refinement. Given a concurrent system Sys and a temporal logic formula ψ, we build the initial spotlight $Spot$. In case that model checking ψ on the corresponding abstraction yields *unknown*, we generate a set $\Pi = \{\pi^1, \ldots, \pi^k\}$ of unconfirmed counterexamples where the number of counterexamples k is a selectable parameter in our heuristic framework. The generated counterexamples hint at a set of refinement candidates $Candidates$. These candidates are also a subset of the vertices of the weighted abstraction dependence graph $WADG$ that we construct for the current abstraction. Based on the $WADG$, we want to define our *heuristic evaluation function* for refinement candidates.

Definition 5.9 (Heuristic Evaluation Function).
Let $WADG = (V, W, D)$ be a weighted abstraction dependence graph representing an abstracted system. Moreover, let $Candidates \subseteq V$ be the corresponding set of refinement candidates derived from unconfirmed counterexamples. A *heuristic evaluation function* is a mapping $h : Candidates \to \mathbb{R}$, where $arg\ max_{v \in Candidates}\ h(v)$ is the heuristically best choice for abstraction refinement.

In our framework, the heuristic evaluation function h is a composition of the functions associated with the weighted abstraction dependence graph. We have defined a number of weight functions for $WADGs$ that quantify aspects which may have a *beneficial* or an *unfavourable* impact on potential refinement steps. Each of these functions is based on a certain heuristic idea. The first function that we define refers to the number of *occurrences* as a candidate:

Heuristic Idea	A candidate that occurs in *multiple* counterexamples, or *multiple times* in a single counterexample is likely a *beneficial* choice for abstraction refinement.
Explanation	Our approach to multiple counterexample-generation yields counterexamples that explicitly cover *different* branches of the spotlight processes' control flow. Adding a candidate to the spotlight that occurs in *several* counterexamples enables us to rule out uncertainty at *multiple parts* of the current abstraction, and thus, gives us a large gain of relevant information.
Function	$occurrence : Candidates \to \mathbb{N}$ where $occurence(v)$ returns the number of occurrences of v as a candidate in the current set of counterexamples Π.

Table 5.1 Weight function *occurence*.

The function *occurrence* characterises a beneficial aspect of potential refinement steps that is apart from dependence. As a dependence-related aspect, we define the *linking factor* of a candidate with the spotlight:

Heuristic Idea	A candidate process that affects a *large number* of spotlight processes is a *beneficial* choice for abstraction refinement. Likewise, a candidate predicate that is *frequently* referenced in spotlight processes is a *beneficial* choice for refinement.
Explanation	Refinement candidates are inherently capable of augmenting the current spotlight abstraction with new definite information. However, the linking between a candidate and the spotlight can range from a loose connection, e.g. a single shared variable, to a tight interweaving with *many mutual dependencies*. Adding a candidate that has a higher *linking factor* with the spotlight, complements the definiteness in the abstraction to a larger extent.
Function	$linkingSpot : Candidates \to \mathbb{N}$ with $linkingSpot(v) := \sum_{v' \in Spot(Proc)} (\omega_1 \cdot D_w(v, v') + \omega_2 \cdot D_w(v', v))$ where $\omega_1, \omega_2 \in \mathbb{N}$ are weight parameters in our heuristic framework.

Table 5.2 Weight function *linkingSpot*.

With *occurrence* and *linkingSpot* we have two weight functions that characterise beneficial aspects of candidates. Both provide us with a measure for the *gain of definite information* by a refinement step. However, abstraction refinement also involves a *gain of complexity*. The *size* of a candidate is one aspect that contributes to the cost of refinement:

Heuristic Idea	The *larger* a refinement candidate, the *more costly* it is to add it to the spotlight.
Explanation	Adding a candidate to the spotlight inherently enlarges the abstract state space. In three-valued abstractions all atomic predicates equally contribute to the overall state space complexity, whereas the complexity induced by processes depends on the number of its control flow locations. In our approach, we internally represent spotlight processes by a set of three-valued predicates over the processes' control flow. This enables us to define a normalised measure for the size of refinement candidates.
Function	$size : V \to \mathbb{N}$ with $size(v) := \begin{cases} 1 & \text{if } v \in Pred \\ min\{k \in \mathbb{N} \mid 3^k \geq \lvert Loc \rvert\} & \text{if } v \in Proc \end{cases}$ where Loc refers to the set of control flow locations of the process v.

Table 5.3 Weight function *size*.

Hence, the size of a predicate is simply one, whereas the size of a process corresponds to the number of predicates that are required to encode its control flow. The *size* function gives us a measure for *immediate costs* of potential refinement steps in terms of the growth of the state space. Besides, this function also helps us to capture *mediate costs* of candidates, induced by *dependencies within the shade*:

Heuristic Idea	A candidate that, directly or transitively, depends on a *large number* of shade processes is a *costly* choice for abstraction refinement.
Explanation	By selecting a certain candidate for refinement, all its dependencies *within* the shade become new dependencies between the spotlight and the shade. Such dependencies that cross the border from the shade to the spotlight are the general sources of uncertainty in the abstraction. Hence, adding a candidate that is affected by a minimal number of shade components is likely a beneficial choice for refinement. However, abstraction refinement is an iterative procedure, and thus, should be performed with foresight. A candidate with a small number of *direct* dependencies may still be involved in a long chain of *transitive* dependencies, e.g. a set of processes that affect one another. By solely focusing on direct dependencies in heuristic decisions there is the risk that iterative refinement gets lost in such a chain, e.g. all processes along the chain are graduallay added to the spotlight, whereas likely beneficial processes with more direct dependencies are entirely ignored. Therefore, we define a function for measuring the shade dependencies of refinement candidates that explicitly considers transitivity. In particular, our function *linkingShade* incorporates the number, size and distance (wrt. transitive dependence) of shade processes that affect a candidate. The contribution of a shade process to the overall linking factor of a refinement candidate decreases proportionally to their distance.
Function	$linkingShade : Candidates \to \mathbb{R}$ with $linkingShade(v) := \sum_{v' \in Shade(Proc) \setminus \{v\}} \frac{size(v')}{distance(v',v)}$ where $distance(v',v)$ returns the length of the shortest directed path from v' to v within the subgraph of the *WADG* induced by the shade. In particular, $distance(v',v)$ yields ∞ if there exists not such path.

Table 5.4 Weight function *linkingShade*.

With *linkingShade* we have a function for measuring costs for refinement candidates, induced by dependencies to *processes*. However, costs for potential refinement steps can also be characterised based on a *predicate-related* aspect:

Heuristic Idea	The more predicates over a certain variable are in the spotlight, the *more redundant* it is to add another predicate over the same variable to the spotlight.
Explanation	By our heuristic idea of avoiding redundancy in the abstraction we counter a problem that frequently occurs when verifying concurrent systems with large-domain loop variables: A loop is completely unrolled by abstraction refinement, i.e. all possible predicates over the loop variable are added to the spotlight – though adding a certain process might already suffice to show termination of the loop. We approach this problem as follows: The more predicates over a distinct variable are in the spotlight, the higher we set the *redundancy*, and hence the cost, of any candidate predicate over the same variable. A process affecting a loop variable eventually will have a better heuristic evaluation than new predicates over this variable, and thus, an unnecessary unwinding of the loop can be avoided.
Function	$redundancy : Candidates \to \mathbb{N}$ with $redundancy(v) := \begin{cases} \lvert\{v' \in Spot(Pred) \mid Ref(v) \cap Ref(v') \neq \emptyset\}\rvert & \text{if } v \in Pred \\ 0 & \text{if } v \in Proc. \end{cases}$

Table 5.5 Weight function *redundancy*.

Each of the previously introduced functions enables us to measure a *specific* characteristic that either has a beneficial, or a costly impact on refinement steps. Our *overall* heuristic evaluation function h now corresponds to a weighted composition of these functions:

$$h(v) :=$$

$$\underbrace{(\omega_1 \cdot occurrence(v) + \omega_2 \cdot linkingSpot(v))}_{benefit(v)}$$

$$-$$

$$\underbrace{(\omega_3 \cdot linkingShade(v) + \omega_4 \cdot size(v) + \omega_5 \cdot redundancy(v))}_{cost(v)}$$

Hence, our heuristic refinement decisions follow from a cost-benefit analysis. The weights $\omega_1, \ldots, \omega_5 \in \mathbb{N}$ are parameters in our framework that allow us to put emphasis on certain aspects in our decisions. For falsifying a universally quantified liveness property it is e.g. advisable to put particular weight on *linkingShade*. The falsification requires to find *one* definite path that refutes the property. Favouring refinement steps that are mainly *independent* from

the rest of the shade facilitates the detection of *straightforward* error traces, i.e. traces that avoid an unnecessarily high degree of interleaving. A more extensive discussion of appropriate parameters for heuristic-guided abstraction refinement, as well as an experimental evaluation of different heuristics, can be found in Chapter 7.

The heuristic evaluation function for candidates completes our framework for abstraction refinement-based verification of concurrent systems. Like other counterexample-guided abstraction refinement approaches e.g. [34], our fully automated technique gradually builds an abstraction that is precise enough for a definite result in verification. However, in contrast to [34] we base our refinement decisions on *multiple* counterexamples and additional *structural information* about the underlying system. The heuristic exploitation of dependencies within the system enables us to distinguish *beneficial* and *costly* refinements, and thus, helps us to guide the refinement procedure in expedient directions. For the verification of concurrent systems this gives us a clear advantage over naive approaches which easily suffer from state explosion caused by unfavourable refinement steps (see Chapter 7). Contrary to CEGAR frameworks based on *counterexample elimination* via lazy abstraction [18], we follow an approach that is more oriented towards the *final result* in verification. However, up to this point our abstraction refinement framework is restricted to the verification of *local* properties, i.e. CTL formulae that refer to a finite set of processes in concurrent systems of fixed size. In the next chapter, we will see that our approach can also be exploited for verifying *global* properties of *parameterised systems* which are composed of an unbounded number of processes. Beforehand, we take a look at related work on abstraction refinement.

5.4 Related Work

Parts of our work introduced in this chapter have already been published in [120]. Besides, our research on heuristic-guided abstraction refinement is connected to other approaches in a number of ways. In this section we summarise and extend our previous references to related works.

Abstraction refinement for verification has received a lot of attention in research. Several approaches in this field are based on boolean predicate abstraction [66, 11] with counterexample-guided abstraction refinement (CEGAR) [34]. Prominent model checking tools like SLAM [12, 15], Blast [18] or CPAchecker [19, 127] build conservative overapproximations of the considered systems. In case that model checking yields a spurious counterexample, the current abstraction is refined by adding new predicates – which are either derived from pre-/postconditions [12, 15], or from Craig interpolants [18, 19] computed for the error path. Craig Interpolation is commonly rated as the more sophisticated approach to refinement. It facilitates to identify particular

parts of the system where the derived predicates are relevant, which can be exploited for applying lazy abstraction [79, 98], i.e. for local refinement. Lazy abstraction involves an on-the-fly integration of the steps *refinement* and *state space exploration*. New predicates are locally added to the abstraction along a spurious error trace, and the exploration of the refined abstraction is continued at previously reached states. However, such an interpolation-based refinement is subject to a number of limitations. It is narrowed to the analysis of finite path prefixes, i.e. to the verification of safety properties, and it is solely compatible with boolean abstractions. – Precondition-based refinement is not affected by these restrictions. The price to pay is that preconditions generally do not hint at local refinements. Our abstraction refinement framework is also based on the computation of preconditions. We refrain from lazy abstraction and take a global view on the verification task. Thus, our refinement procedure is geared towards the final verification result – and not on the elimination of a single counterexample, which is pursued in [34, 12, 18, 79]. Another difference between these classical CEGAR frameworks and our approach is that we operate in a three-valued domain, and thus, unify over- and underapproximation in one abstraction. This cumbers an on-the-fly approach to refinement and exploration, but enables us to support the verification of full CTL properties.

A framework for *three-valued abstraction refinement* in model checking has first been introduced by Grumberg et al. [68, 69]. Their approach to abstraction is similar to ours, i.e. their abstract models preserve full branching time properties. Unlike us, they apply *game-based* model checking – which yields *game graphs* rather than counterexamples. Nevertheless, these graphs are likewise exploited for (global) abstraction refinement, since they hint at abstract states that should be splitted in order to eliminate the cause of an indefinite result. The work of [68, 69] has been further enhanced by local refinement (i.e. lazy abstraction) and a number of suggested refinement strategies by Fecher and Shoham in [61]. However, the entire framework ([68, 69, 61]) is of theoretical nature; so far there exists no implementation. A first approach to spotlight abstraction refinement was proposed by Toben in [121]. The presented technique is not based on predicate abstraction. Hence, refinement is limited to adding processes to the spotlight. The three-valued CEGAR framework of Schrieb et al. [112], in many respects the basis of our approach, originally introduced the idea of refinement in terms of adding predicates *and* processes to the abstraction.

The concept of *multiple counterexample-generation* for abstraction refinement has also been considered in a number of works. Esparza et al. [60] present a boolean abstraction refinement technique for verifying safety properties of sequential programs. In each iteration a directed acyclic graph is computed that represents *all* error traces in the current abstraction. If none of these traces is feasible for the concrete program then all of them are excluded via interpolation-based refinement. – Our approach is not limited to safety properties, and furthermore focuses on concurrent systems. These two charac-

teristics substantially increase the general complexity of the abstraction and of corresponding counterexamples. Thus, we refrain from generating an entire graph of all abstract counterexamples. Instead, we permit the generation of multiple counterexamples only for particular branches that can be exploited for region summarisation (compare Section 4.2.3). The number of actually generated counterexamples is additionally bounded by a parameter of our heuristic framework. Another approach to iterative abstraction refinement guided by multiple counterexamples is proposed by Glusman et al. [63]. Their framework is tailored to the verification of safety properties of hardware designs, and their abstractions are based on *variable hiding*, i.e. pruning parts of the design logic. In each iteration all abstract counterexamples of a given length are generated via SAT-based bounded model checking. Refinement is performed by freeing parts of the hidden logic that are relevant for the elimination of spurious error traces. This step is guided by a heuristic which categorises the pruned hardware components into *strong*, *conditional* and *irrelevant* with regard to the refutation of the set of spurious counterexamples. Similar to our approach, [63] aim at detecting refinements that are multilaterally valuable for the abstraction – not just for the elimination of a single error trace. However, their focus is on the verification of hardware designs and they are not building on a predicate abstraction framework.

Heuristic-guided abstraction refinement in the context of hardware verification is also considered in [75, 76]. Therein, heuristics are used for finding an approximative solution for the NP-hard *minimal state separation problem*, i.e. for detecting a minimal set of currently hidden variables that are capable of separating abstract states such that a single spurious counterexample is eliminated. The variable selection is heuristically guided in terms of maximising the number of pairs of abstract states that are newly separated. A similar technique has been presented by Kurshan [91] who uses a heuristic for freeing hidden variables based on a variable dependence analysis. A hybrid approach to counterexample-guided abstraction refinement that combines predicate abstraction and variable hiding is proposed by Wang et al. [125]. Their heuristics for selecting new predicates or hidden variables for refinement are based on a static analysis of the hardware design under consideration. The latter two approaches [91, 125] are related to our work in the sense that they also base their refinement decisions on a structural analysis of the considered system. However, they focus on variable dependencies within a sequential hardware circuit rather than on inter-process dependencies in a concurrent system.

Heuristic guidance in the context of verifying *concurrent systems* has also been used by Tan et al. [118]. The authors present an extension of the verification tool FLAVERS [104, 42]. Heuristics for selecting constraint automata that rule out infeasible interleavings in a FLAVERS model of a concurrent system are proposed. FLAVERS models are not based on predicate abstraction but on control flow graphs enriched with constraint automata. The applied heuristics exploit the structure of the considered system, the property to be

checked, and additional constraints that have to be selected manually in advance. Hence, their approach is neither fully automatic nor integrated into an iterative, counterexample-based refinement framework. Another approach to abstraction refinement for concurrent systems related to our work is that of Gupta et al. [70]. The authors propose a compositional verification technique for safety properties, based on rely-guarantee reasoning [106, 85]. Each process of a concurrent system is verified in isolation, whereas the behaviour of the remaining system is summarised by an overapproximating environment assumption. In case of a spurious counterexample, the environment assumption is automatically refined. The summarisation of the environment is very similar to our shade component. However, we refrain from a compositional approach to verification that requires to consider every single process of the overall system. Instead, we aim at heuristically discovering a minimal set of processes that is sufficiently large for a definite answer in verification.

The idea of using heuristics in verification has also been considered in a different context than abstraction refinement. *Directed model checking* [56, 55, 81] is a common approach to counterexample generation in temporal logic verification. Based on heuristic search strategies the state space of the considered system is explored in order to find counterexamples of minimal length. The intention behind this approach is to produce small error traces that are easy to understand and to fix by a user. In contrast, we aim at minimising the size of the final abstraction on which a definite result in verification can be obtained. Shortness of counterexamples is not of major importance in our framework, since error paths are automatically processed. In fact, minimal counterexamples are even adverse for getting a variety of refinement candidates.

Finally, our work is related to *dependence analysis* techniques for concurrent systems. Methods for constructing dependence graphs of concurrent systems are presented in [31, 111, 100]. These approaches focus on dependencies between single operations, whereas we perform an inter-process dependency analysis. Hence, our analysis does not expose dependencies on the level of operations but is significantly more cost-efficient – which is in line with our notion of applying heuristics in the sense of exploiting *easily* accessible information.

Chapter 6
Spotlight Abstraction for Parameterised Verification

So far, our abstraction refinement framework for concurrent systems is tailored to the verification of local requirements that refer to a small set of processes. Validating global properties like *"each process will continuously proceed"* is generally possible. However, it requires to take the entire system into the spotlight – which, of course, contradicts the fundamental idea of spotlight abstraction. Another limitation of our current framework becomes evident when we consider parameterised systems (compare Section 3.2), i.e. parallel compositions of unbounded numbers of processes. The verification of local requirements is basically feasible for such a system, although our refinement loop might not terminate due to an unbounded number of processes in the shade. – But verifying global requirements would confront us with a major problem: The spotlight has to comprise an unlimited number of processes. However, this issue does not stem from any particular weakness of our approach. Parameterised verification, i.e. checking global properties of systems with an arbitrary number of processes, is undecidable in general [7].

Nevertheless, a number of techniques have been developed that successfully address the parameterised verification problem. Common approaches refrain from completeness and impose certain restrictions on the system under consideration. These restrictions typically concern *symmetry*, i.e. the recurrence of similar structures. Many real-life parameterised systems are inherently symmetric. For instance, network protocols are usually designed for an arbitrary number of clients that are either homogeneous, or that can at least be divided into a finite number of classes of homogeneous clients. *Symmetry reduction* methods [58, 105] exploit such homogeneity in parameterised systems. They map the unbounded state spaces to finite representations that can be verified.

In this chapter, we show that symmetry reduction can be effectively integrated into our framework for abstraction refinement. The combination of spotlight abstraction with symmetry arguments enables us to extend our approach towards the efficient verification of parameterised systems. We start this chapter with the introduction of the foundations of parameterised verification.

6.1 Parameterised Verification

As a motivating example for parameterised verifcation, we want to consider a network protocol for mutual exclusion. The protocol has to guarantee exclusive access to a shared resource – regardless of the number of clients in the network. This problem can be straightforwardly transferred to the field of parameterised systems. In Section 3.2 we proposed the following solution by means of a semaphore:

$$y : \textbf{semaphore where } y = 1$$

$$\|_{i \in PID_N}\ Proc_i :: \begin{bmatrix} 1: & \textbf{loop forever do} \\ & \begin{bmatrix} 2: & \textbf{non-critical} \\ 3: & \textbf{acquire}(y,1) \\ 4: & \textbf{critical} \\ 5: & \textbf{release}(y,1) \end{bmatrix} \end{bmatrix}$$

Fig. 6.1 Parameterised system $Sys = \|_{i \in PID_N}\ Proc_i$ over $Var = Var_s = \{y\}$ where PID_N is a set of process indices with a parameterised size $N \in \mathbb{N}$.

Note that this fully symmetric system (compare Definition 3.3) is *parameterised* with regard to the number of processes. Hence, Sys represents a mutual exclusion mechanism for an *arbitrary* number of replicated clients. We use a capital N to refer to the parameter itself, and a small n to generally refer to the fixed size of an instantiation. One concrete instantiation Sys_4 with four processes, i.e. $PID_4 = \{1,2,3,4\}$, is given in the figure below.

$$y : \textbf{semaphore where } y = 1$$

$$Proc_1 :: \begin{bmatrix} 1: & \textbf{loop forever do} \\ & \begin{bmatrix} 2: & \textbf{non-critical} \\ 3: & \textbf{acquire}(y,1) \\ 4: & \textbf{critical} \\ 5: & \textbf{release}(y,1) \end{bmatrix} \end{bmatrix} \| \; Proc_2 :: \begin{bmatrix} 1: & \textbf{loop forever do} \\ & \begin{bmatrix} 2: & \textbf{non-critical} \\ 3: & \textbf{acquire}(y,1) \\ 4: & \textbf{critical} \\ 5: & \textbf{release}(y,1) \end{bmatrix} \end{bmatrix}$$

$$\| \; Proc_3 :: \begin{bmatrix} 1: & \textbf{loop forever do} \\ & \begin{bmatrix} 2: & \textbf{non-critical} \\ 3: & \textbf{acquire}(y,1) \\ 4: & \textbf{critical} \\ 5: & \textbf{release}(y,1) \end{bmatrix} \end{bmatrix} \| \; Proc_4 :: \begin{bmatrix} 1: & \textbf{loop forever do} \\ & \begin{bmatrix} 2: & \textbf{non-critical} \\ 3: & \textbf{acquire}(y,1) \\ 4: & \textbf{critical} \\ 5: & \textbf{release}(y,1) \end{bmatrix} \end{bmatrix}$$

Fig. 6.2 Instantiation Sys_4 of the parameterised system Sys with $PID_4 = \{1,2,3,4\}$.

We now want to apply our verification framework and check whether the local mutual exclusion property $\mathbf{AG}\neg(pc_1 = 4 \wedge pc_2 = 4)$ holds for this instantiation. Remember that in our parameterised systems the local variables of each process are initialised with the same values, i.e. in our example all processes start their computation at control location 1. Taking $Proc_1$ and $Proc_2$ into the

spotlight, together with two predicates over the semaphore y, and leaving $Proc_3$ and $Proc_4$ in the shade is sufficient in order to prove that the first two processes will be never in the critical section at the same time. However, this result tells us nothing about *global* mutual exclusion. We have to check whether *all* pairs of processes will never be simultaneously in the critical section. This can be formalised as follows:

$$\Psi := \bigwedge_{\langle i_1, i_2 \rangle \in [PID_4]^2_{\neq}} \mathbf{AG} \neg (pc_{i_1} = 4 \wedge pc_{i_2} = 4)$$

with

$$[PID_4]^2_{\neq} := \{ \langle i_1, i_2 \rangle \mid i_1, i_2 \in PID_4,\ i_1 \neq i_2 \},$$

i.e. $[PID_4]^2_{\neq}$ denotes the second cartesian power of the set PID_4 where all tuples $\langle i_1, i_2 \rangle \in [PID_4]^2_{\neq}$ consist of *pairwise different* process identifiers from PID_4. The formula Ψ refers to the entire instantiation of the parameterised system, and thus, its validation requires to draw all processes into the spotlight. This might be practicable for very small instantiations, but certainly not for the scales of real-life networks. However, a more crucial issue is the fact that the correctness of one instantiation does not allow us to draw any conclusions about the correctness of the entire mutual exclusion mechanism. A prominent example for a parameterised system whose correctness depends on the size of its instantiation is Peterson's algorithm for mutual exclusion [107]. The mechanism guarantees mutual exclusion for two processes but not for larger instantiations. Hence, in order to prove a property of a parameterised system *all* possible instantiations have to be regarded. For our running example we have to validate the following verification task

$$\forall n \geq 2 \ :\ K(Sys_n), s_0 \ \models \bigwedge_{\langle i_1, i_2 \rangle \in [PID_n]^2_{\neq}} \mathbf{AG} \neg (pc_{i_1} = 4 \wedge pc_{i_2} = 4),$$

i.e. we check whether for all instantiations of size greater or equal 2 (the number of different process variables in the requirement) the mutual exclusion property holds for every pair of processes. Here $K(Sys_n)$ denotes the Kripke structure corresponding to the instantiation Sys_n, where $n \in \mathbb{N}$ is the size of the instantiation and s_0 is the state of $K(Sys_n)$ that represents the initial configuration of Sys_n. Apparently, a straightforward approach to this verification task requires an infinite number of model checking runs, and thus, will not terminate. Even our current framework for abstraction refinement cannot yield any improvement here. The general formulations of global temporal logic formulae and the parameterised verification problem for fully symmetric systems are given in the subsequent definitions.

Definition 6.1 (Global CTL Formulae over Fully Symmetric Systems). Let $Sys = \|_{i \in PID_N} Proc_i$ be a fully symmetric parameterised system. Moreover, let $\psi(i_1, \ldots, i_d)$ be a parameterised CTL formula with reference to variables for process identifiers from PID_N. Then the corresponding *global CTL formula* is

$$\Psi := \bigwedge_{\langle i_1,\ldots,i_d\rangle \in [PID_N]^d_{\neq}} \psi(i_1,\ldots,i_d)$$

where

$$[PID_N]^d_{\neq} := \{\langle i_1,\ldots,i_d\rangle \mid i_1,\ldots,i_d \in PID_N,\ i_j \neq i_k \text{ for all } 1 \leq j,k \leq d, j \leq k\}.$$

Thus, $[PID_N]^d_{\neq}$ denotes the d-th cartesian power of the set PID_N where all d-tuples $\langle i_1,\ldots,i_d\rangle \in [PID_N]^d_{\neq}$ consist of pairwise different process identifiers from PID_N. Consequently, the global formula Ψ is a conjunction over all possible combinations of d pairwise different processes, where each clause of Ψ corresponds to a local CTL formula that refers to d specific processes.

Definition 6.2 (Parameterised Verification of Fully Symmetric Systems). Let $Sys = \|_{i\in PID_N} Proc_i$ be a fully symmetric parameterised system and let $\Psi = \bigwedge_{\langle i_1,\ldots,i_d\rangle \in [PID_N]^d_{\neq}} \psi(i_1,\ldots,i_d)$ be a global CTL formula over Sys. Then the corresponding *parameterised verification problem* is

$$\forall n \geq d \ :\ K(Sys_n), s_0 \models \bigwedge_{\langle i_1,\ldots,i_d\rangle \in [PID_n]^d_{\neq}} \psi(i_1,\ldots,i_d)$$

where $K(Sys_n)$ is a Kripke structure corresponding to Sys_n and s_0 is the state of $K(Sys_n)$ that represents the initial configuration of Sys_n.

Hence, parameterised verification presents us a generally undecidable problem. Nevertheless, in the next section we will see that symmetry allows us to reduce the global property to be checked to a local one. The verification of local properties however can be efficiently performed based on spotlight abstraction. In Section 6.3 we then will show that the combination of symmetry reduction and spotlight abstraction enables us to transfer definite verification results obtained on abstract models to the entire parameterised system.

6.2 Symmetry Reduction

In the first step of our approach to parameterised verification we show that, based on symmetry arguments, checking global system requirements can be reduced to checking local requirements. We assume that we have given an arbitrary but fixed instantiation $Sys_n = \|_{i\in PID_n} Proc_i$ over $Var = Var_s \cup (Var_l \times PID_n)$ of a fully symmetric system and a global CTL formula $\Psi = \bigwedge_{\langle i_1,\ldots,i_d\rangle \in [PID_n]^d_{\neq}} \psi(i_1,\ldots,i_d)$ over Sys_n. Ψ refers to all possible combinations of d pairwise different processes of Sys_n. These combinations can be characterised by permutations on process identifiers:

Definition 6.3 (Process Permutation).
Let $Sys_n = \|_{i\in PID_n} Proc_i$ be an instantiation of a fully symmetric parameterised

system. Then a corresponding *process permutation* is a bijective function

$$\sigma : PID_n \to PID_n.$$

Hence, a process permutation corresponds to a reordering of the identical processes in a fully symmetric system. For instance, applying the permutation $\sigma = \{(1,2),(2,1),(3,3),(4,4)\}$ to the instantiation Sys_4 in Figure 6.2 interchanges the roles of the processes $Proc_1$ and $Proc_2$, while $Proc_3$ and $Proc_4$ are not affected by this permutation.

Subsequently we will show that the validity of global temporal logic properties of the form $\bigwedge_{\langle i_1,\ldots,i_d\rangle \in [PID_n]^d_{\neq}} \psi(i_1,\ldots,i_d)$ is preserved under permutation. Here we follow standard approaches for symmetry reduction [58, 105, 39]. However, our symmetries concern only process identifiers, but not data variables. – First, we require a computational model corresponding to a given system instantiation $Sys_n = \|_{i \in PID_n} Proc_i$. According to Definition 3.7, Sys_n can be represented as a Kripke structure $K = (S,R,L,\mathbb{F})$ over a set of atomic predicates AP. Without loss of generality we define AP as

$$\begin{aligned} &\{(x = val) \mid x \in Var_s \ \wedge \ val \in dom(x)\} \\ \cup\ &\{((x,i) = val) \mid (x,i) \in Var_l \times PID_n \ \wedge \ val \in dom(x)\} \\ \cup\ &\{(pc_i = j) \mid i \in PID_n \ \wedge \ j \in Loc\} \end{aligned}$$

where $dom(x)$ denotes the domain of a variable $x \in Var$ and Loc denotes the set of control locations of processes in Sys.

Next, we lift process permutations to variables of a fully symmetric system and to states of a corresponding Kripke structure.

Definition 6.4 (Process Permutations for Variables and States).
Let $Sys_n = \|_{i \in PID_n} Proc_i$ over $Var = Var_s \cup (Var_l \times PID_n)$ be an instantiation of a fully symmetric parameterised system. Moreover, let $K = (S,R,L,\mathbb{F})$ over AP be a corresponding Kripke structure. Then a process permutation $\sigma : PID_n \to PID_n$ can be extended to variables of Sys_n and states $s \in S$ as follows:

- for shared variables $x \in Var_s$:
  $\sigma(x) = x$ and $\sigma(s)(x) = s(x)$,
- for local variables $(x,i) \in Var_l \times i$, $i \in PID_n$:
  $\sigma(x,i) = (x,\sigma(i))$ and $\sigma(s)(x,i) = s(x,\sigma(i))$,
- for program counters pc_i, $i \in PID_n$:
  $\sigma(pc_i) = (pc_{\sigma(i)})$ and $\sigma(s)(pc_i) = s(pc_{\sigma(i)})$.

Thus, the valuations of shared variables are not affected by permutation, whereas the values of local variables (x,i) associated with some process $Proc_i$ are substituted by the values of the corresponding variables $(x,\sigma(i))$ of the process $Proc_{\sigma(i)}$. By recursion, we can lift permutations to any expression over

the system variables, in particular to atomic predicates in AP. We get $\sigma(x = val) = (x = val)$, $\sigma((x,i) = val) = ((x,\sigma(i)) = val)$ and $\sigma(pc_i = j) = (pc_{\sigma(i)} = j)$.

In order to exploit process permutations for temporal logic verification, they have to preserve the semantics of Kripke structures in the sense of symmetry:

Definition 6.5 (Symmetry).
Let $Sys_n = \|_{i \in PID_n} Proc_i$ be an instantiation of a fully symmetric parameterised system and let $K = (S, R, L, \mathbb{F})$ over AP be a corresponding Kripke structure. A process permutation $\sigma : PID_n \to PID_n$ is a *symmetry* for K if the following conditions are met

1. for all process indices $i \in PID_n$ and for all pairs of states $s, s' \in S$:
   $R_i(s,s') \;\Leftrightarrow\; R_{\sigma(i)}(\sigma(s), \sigma(s'))$,

2. for all states $s \in S$ and for all atomic predicates $p \in AP$:
   $L(s,p) \;\Leftrightarrow\; L(\sigma(s), \sigma(p))$,

3. for all process indices $i \in PID_n$ and for all pairs of states $s, s' \in S$:
   $(s,s') \in F_i \;\Leftrightarrow\; (\sigma(s), \sigma(s')) \in F_{\sigma(i)}$.

For fully symmetric systems we thus get the following lemma:

Lemma 6.1.
On Kripke structures corresponding to instantiations of fully symmetric systems, all process permutations are symmetries.

Proof (Lemma 6.1).
We start with condition 1 of Definition 6.4 and show that permutations preserve the transition relation. Let Sys_n, K and σ be defined as in Definition 6.4. Then for any process identifier $i \in PID_n$ and for any two states $s, s' \in S$ we have that:

$R_i(s,s')$

$\overset{(1)}{\Rightarrow}$ $R_i(\langle l, s_v\rangle, \langle l', s'_v\rangle)$
where l, l' denote the location parts and s_v, s'_v the system variable parts of s resp. s'

$\overset{(2)}{\Rightarrow}$ there is a basic operation $bop_i \;=\; assume(e) : x_1 := e_1, \ldots, x_m := e_m$
with $x_1, \ldots, x_m \in Var_i = Var_s \cup (Var_l \times i)$
and $e, e_1, \ldots, e_m$ expressions over Var_i, such that
$\delta\left(l, bop_i, i, l'\right) \wedge s_v(e) \wedge s'_v(x_1) = s_v(e_1) \wedge \ldots \wedge s'_v(x_m) = s_v(e_m)$

$\overset{(3)}{\Rightarrow}$ $\delta\left(l, bop_i, i, l'\right) \wedge s(e) \wedge s'(x_1) = s(e_1) \wedge \ldots \wedge s'(x_m) = s(e_m)$

$\overset{(4)}{\Rightarrow}$ $\delta_i\left(l_i, bop_i, l'_i\right) \wedge s(e) \wedge s'(x_1) = s(e_1) \wedge \ldots \wedge s'(x_m) = s(e_m)$

$$\stackrel{(5)}{\Rightarrow} \delta_{\sigma(i)}\left(l_{\sigma(i)}, bop_{\sigma(i)}, l'_{\sigma(i)}\right) \wedge$$
$$\sigma(s)(\sigma(e)) \wedge \sigma(s')(\sigma(x_1)) = \sigma(s)(\sigma(e_1)) \wedge \ldots \wedge \sigma(s')(\sigma(x_m)) = \sigma(s)(\sigma(e_m))$$
$$\text{where } bop_{\sigma(i)} = assume(\sigma(e)) : \sigma(x_1) := \sigma(e_1), \ldots, \sigma(x_m) := \sigma(e_m)$$
$$\stackrel{(6)}{\Rightarrow} R_{\sigma(i)}(\sigma(s), \sigma(s'))$$

In step (1) we split up the states of the Kripke structure into location parts and system variable parts. Step (2) follows from Definition 3.7, the transformation of a concurrent system into a Kripke structure, which we can also perform backwards. Definition 3.3 (fully symmetric systems) additionally guarantees us that the operation bop_i associated with $R_i(s,s')$ is solely defined over variables from $Var_s \cup (Var_l \times i)$. Step (3) is based on the fact that $x_1, \ldots, x_m$ and $e, e_1, \ldots, e_m$ refer to system variables only, and that s_v is included in s, and s'_v is included in s'. Step (4) directly follows from Definition 3.6 (concurrent systems as control flow graphs). For step (5), we combine Definition 3.3 (fully symmetric systems), 6.2 (process permutations) and 6.3 (process permutations for variables and states). The extension of σ to expressions $e, e_1, \ldots, e_m$ over $Var_s \cup (Var_l \times i)$ is a straightforward generalisation of Definition 6.3. The final step is again based on Definition 3.7. The reverse direction is proved analogously.

Second, we prove that process permutations also preserve the labelling function. The argumentation is based on Definition 3.7 (concurrent systems as Kripke structures) and 6.3 (process permutations for variables and states), and the fact that permutations can be recursively applied to any expression over system variables and program counters. We can distinguish the following cases:

- Let $p \in AP$ be of the form $(x = val)$:

$$\begin{aligned} & L(s, (x = val)) \\ \Leftrightarrow\ & s(x = val) \\ \Leftrightarrow\ & \sigma(s)(x = val) \\ \Leftrightarrow\ & L(\sigma(s), (x = val)) \\ \Leftrightarrow\ & L(\sigma(s), \sigma(x = val)) \end{aligned}$$

- Let $p \in AP$ be of the form $((x,i) = val)$:

$$\begin{aligned} & L(s, ((x,i) = val)) \\ \Leftrightarrow\ & s((x,i) = val) \\ \Leftrightarrow\ & \sigma(s)((x, \sigma(i)) = val) \\ \Leftrightarrow\ & L(\sigma(s), ((x, \sigma(i)) = val)) \\ \Leftrightarrow\ & L(\sigma(s), \sigma((x,i) = val)) \end{aligned}$$

- Let $p \in AP$ be of the form $(pc_i = j)$:

$$\begin{aligned} & L(s,(pc_i = j)) \\ \Leftrightarrow\ & s(pc_i = j) \\ \Leftrightarrow\ & \sigma(s)(pc_{\sigma(i)} = j) \\ \Leftrightarrow\ & L(\sigma(s),(pc_{\sigma(i)} = j)) \\ \Leftrightarrow\ & L(\sigma(s),\sigma((pc_i = j))) \end{aligned}$$

Finally, we show that the set of fairness constraints is preserved under permutation as well. The argumentation is based on Definition 3.7 (concurrent systems as Kripke structures) and on the already proved validity of condition 1 of Definition 6.4 for fully symmetric systems.

$$\begin{aligned} & (s,s') \in F_i \\ \Leftrightarrow\ & R_i(s,s') \neq false \\ \Leftrightarrow\ & R_{\sigma(i)}(\sigma(s),\sigma(s')) \neq false \\ \Leftrightarrow\ & (\sigma(s),\sigma(s')) \in F_{\sigma(i)} \end{aligned}$$

□

From this lemma we can immediately deduce the following corollary, which reveals that fair paths are preserved under permutation.

Corollary 6.1.
Let σ be a symmetry for a Kripke structure $K = (S,R,L,\mathbb{F})$ over a set of atomic predicates AP. Moreover, let $\pi = s_0 s_1 s_2 \ldots$ be a fair path of K, i.e. $\pi \in \Pi_{s_0}^{fair}$. Then $\sigma(\pi) = \sigma(s_0)\sigma(s_1)\sigma(s_2)\ldots$ is a fair path of K as well, i.e. $\sigma(\pi) \in \Pi_{\sigma(s_0)}^{fair}$, with $L(s_i,p) = L(\sigma(s_i),\sigma(p))$ for all atomic predicates $p \in AP$ and $i \in \mathbb{N}$. We then say that π and $\sigma(\pi)$ are symmetric paths.

Hence, given a Kripke structure $K = (S,R,L,\mathbb{F})$ corresponding to an instantiation of a fully symmetric system $Sys_n = \|_{i \in PID_n} Proc_i$, then for each fair path π and each permutation σ on PID_n there exists a fair symmetric path $\sigma(\pi)$ in K. Now let us imagine that the path π is a counterexample (or a witness) for some local temporal logic property $\psi(pid_1,\ldots,pid_d)$ with $pid_1,\ldots,pid_d \in PID_n$, i.e. a property that refers to d specific processes. According to the following theorem we can deduce that the symmetric path $\sigma(\pi)$ is a counterexample (witness) for the permuted property $\psi(\sigma(pid_1),\ldots,\sigma(pid_d))$.

Theorem 6.1.
Let $Sys_n = \|_{i \in PID_n} Proc_i$ be an instantiation of a fully symmetric system, $K = (S,R,L,\mathbb{F})$ be the corresponding Kripke structure over a set of atomic predicates AP, and σ be a process permutation, i.e. a symmetry for K. Moreover, let $\psi(pid_1,\ldots,pid_d)$ be a CTL formula over AP with $pid_1,\ldots,pid_d \in PID_n$, and let $s \in S$ be a state of K. Then

$$[K,s \models \psi(pid_1,\ldots,pid_d)] \quad \Leftrightarrow \quad [K,\sigma(s) \models \psi(\sigma(pid_1),\ldots,\sigma(pid_d))].$$

Proof (Theorem 6.1).
The proof proceeds by induction on the structure of the CTL formula ψ. The argumentation is based on the fair three-valued CTL semantics (Definition 2.10) and on Corollary 6.1. We consider the following cases:

- Let ψ be of the form p where $p \in AP$:

$$\begin{aligned} & [K,s \models p] \\ \Leftrightarrow & \bigvee_{\pi \in \Pi_s^{fair}} L(\pi_0,p) \\ \Leftrightarrow & \bigvee_{\sigma(\pi) \in \Pi_{\sigma(s)}^{fair}} L(\sigma(\pi_0),\sigma(p)) \\ \Leftrightarrow & \ [K,\sigma(s) \models \sigma(p)] \end{aligned}$$

- Let ψ be of the form $\neg\psi'$ where ψ' is a CTL formula:

$$\begin{aligned} & [K,s \models \neg\psi'] \\ \Leftrightarrow & \bigvee_{\pi \in \Pi_s^{fair}} \neg\left[K,\pi_0 \models \psi'\right] \\ \Leftrightarrow & \bigvee_{\sigma(\pi) \in \Pi_{\sigma(s)}^{fair}} \neg\left[K,\sigma(\pi_0) \models \sigma(\psi')\right] \\ \Leftrightarrow & \ [K,\sigma(s) \models \sigma(\neg\psi')] \end{aligned}$$

- Let ψ be of the form $\psi_1 \vee \psi_2$ where ψ_1, ψ_2 are CTL formulae:

$$\begin{aligned} & [K,s \models \psi_1 \vee \psi_2] \\ \Leftrightarrow & \bigvee_{\pi \in \Pi_s^{fair}} [K,\pi_0 \models \psi_1] \ \vee \ [K,\pi_0 \models \psi_2] \\ \Leftrightarrow & \bigvee_{\sigma(\pi) \in \Pi_{\sigma(s)}^{fair}} [K,\sigma(\pi_0) \models \sigma(\psi_1)] \ \vee \ [K,\sigma(\pi_0) \models \sigma(\psi_2)] \\ \Leftrightarrow & \ [K,\sigma(s) \models \sigma(\psi_1 \vee \psi_2)] \end{aligned}$$

- Let ψ be of the form $\mathbf{EX}\psi'$ where ψ' is a CTL formula:

$$
\begin{aligned}
& [K, s \models \mathbf{EX}\psi'] \\
\Leftrightarrow \quad & \bigvee_{\pi \in \Pi_s^{fair}} R(\pi_0, \pi_1) \wedge \left[K, \pi_1 \models \psi'\right] \\
\Leftrightarrow \quad & \bigvee_{\sigma(\pi) \in \Pi_{\sigma(s)}^{fair}} R(\sigma(\pi_0), \sigma(\pi_1)) \wedge \left[K, \sigma(\pi_1) \models \sigma(\psi')\right] \\
\Leftrightarrow \quad & [K, \sigma(s) \models \sigma(\mathbf{EX}\psi')]
\end{aligned}
$$

- Let ψ be of the form $\mathbf{EG}\psi'$ where ψ' is a CTL formula:

$$
\begin{aligned}
& [K, s \models \mathbf{EG}\psi'] \\
\Leftrightarrow \quad & \bigvee_{\pi \in \Pi_s^{fair}} \bigwedge_{i \in \mathbb{N}} \left(R(\pi_i, \pi_{i+1}) \wedge \left[K, \pi_i \models \psi'\right]\right) \\
\Leftrightarrow \quad & \bigvee_{\sigma(\pi) \in \Pi_{\sigma(s)}^{fair}} \bigwedge_{i \in \mathbb{N}} \left(R(\sigma(\pi_i), \sigma(\pi_{i+1})) \wedge \left[K, \sigma(\pi_i) \models \sigma(\psi')\right]\right) \\
\Leftrightarrow \quad & [K, \sigma(s) \models \sigma(\mathbf{EG}\psi')]
\end{aligned}
$$

- Let ψ be of the form $\mathbf{E}[\psi_1 \mathbf{U} \psi_2]$ where ψ_1, ψ_2 are CTL formulae:

$$
\begin{aligned}
& [K, s \models \mathbf{E}[\psi_1 \mathbf{U} \psi_2]] \\
\Leftrightarrow \quad & \bigvee_{\pi \in \Pi_s^{fair}} \bigvee_{i \in \mathbb{N}} \left([K, \pi_i \models \psi_2] \wedge \bigwedge_{0 \leq j < i} \left(R(\pi_j, \pi_{j+1}) \wedge [K, \pi_j \models \psi_1]\right) \right) \\
\Leftrightarrow \quad & \bigvee_{\sigma(\pi) \in \Pi_{\sigma(s)}^{fair}} \bigvee_{i \in \mathbb{N}} \\
& \left([K, \sigma(\pi_i) \models \sigma(\psi_2)] \wedge \bigwedge_{0 \leq j < i} \left(R(\sigma(\pi_j), \sigma(\pi_{j+1})) \wedge [K, \sigma(\pi_j) \models \sigma(\psi_1)]\right) \right) \\
\Leftrightarrow \quad & [K, \sigma(s) \models \sigma(\mathbf{E}[\psi_1 \mathbf{U} \psi_2])]
\end{aligned}
$$

In the remaining cases, the correctness results from the equivalences for CTL formulae (compare Section 2.1).
□

Now we want to see how we can exploit this theorem for our approach to parameterised verification. Again, we assume that we have given a fixed instantiation $Sys_n = \|_{i \in PID_n} Proc_i$ of a fully symmetric system and a global CTL formula $\Psi = \bigwedge_{\langle i_1, \ldots, i_d \rangle \in [PID_n]^d_{\neq}} \psi(i_1, \ldots, i_d)$ over Sys_n. In fact, Ψ is a conjunction over all possible combinations of d pairwise different processes of Sys_n, where each clause of Ψ corresponds to a *local* CTL formula that refers to d *specific* processes. Hence, based on Definition 6.2 we can reformulate Ψ as follows:

$$\Psi \;\Leftrightarrow\; \bigwedge_{\sigma\in\Sigma} \psi(\sigma(pid_1),\ldots,\sigma(pid_d))$$

where Σ is the set of all process permutations over PID_n, and $pid_1,\ldots,pid_d \in PID_n$ are arbitrary, pairwise different process identifiers.

In order to check whether this temporal logic property holds for Sys_n we construct a corresponding Kripke structure $K = (S,R,L,\mathbb{F})$ over a set of atomic predicates AP. Remember that initially all processes of a fully symmetric system are in the same local state. Thus, for a state $s_0 \in S$ that represents the initial configuration of Sys we have that $\sigma(s_0) = s_0$ for all process permutations $\sigma \in \Sigma$. Now we just consider one clause of Ψ and check $[K,s_0 \models \psi(pid_1,\ldots,pid_d)]$. According to Theorem 6.1, the obtained result can be transferred to *any* symmetric model checking task $[K,s_0 \models \psi(\sigma(pid_1),\ldots,\sigma(pid_d))]$ with $\sigma \in \Sigma$. Consequently, we get the same result for the corresponding *global* task $[K,s_0 \models \bigwedge_{\sigma\in\Sigma} \psi(\sigma(pid_1),\ldots,\sigma(pid_d))]$. The subsequent corollary from Theorem 6.1 summarises the concept of symmetry reduction.

Corollary 6.2.
Let $Sys_n = \|_{i\in PID_n} Proc_i$ be an instantiation of a fully symmetric system, $K = (S,R,L,\mathbb{F})$ be the corresponding Kripke structure over a set of atomic predicates AP, and $s_0 \in S$ be the state of K that represents the initial configuration of Sys_n. Moreover, let $\Psi = \bigwedge_{\langle i_1,\ldots,i_d\rangle \in [PID_n]^d_{\neq}} \psi(i_1,\ldots,i_d)$ be a global CTL formula over AP, and $pid_1,\ldots,pid_d \in PID_n$ pairwise different process identifiers. Then

$$[K,s_0 \models \Psi] \quad\Leftrightarrow\quad [K,s_0 \models \psi(pid_1,\ldots,pid_d)].$$

Hence, symmetry allows us to reduce global verification tasks to local ones. In order to see how we can profit from this result in our approach to verification, we consider again our running example. The instantiation Sys_4 of the parameterised system in Figure 6.2 consists of four identical processes. We want to verify that never more than one process will be in the critical section at the same time, i.e. whether $\bigwedge_{\langle i_1,i_2\rangle \in [PID_4]^2_{\neq}} \mathbf{AG}\neg(pc_{i_1} = 4 \wedge pc_{i_2} = 4)$ holds for Sys_4. Directly checking this global formula would require to take the entire instantiation into the spotlight. Instead, we consider just one clause of the original formula: The local property $\mathbf{AG}\neg(pc_1 = 4 \wedge pc_2 = 4)$ can be validated on an abstraction with only $Proc_1$ and $Proc_2$ in the spotlight – and based on our symmetry arguments we can deduce that mutual exclusion holds *globally*. However, up to this point we have still no solution for the general parameterised verification problem, i.e. whether a global property holds for *all* instantiations of a parameterised system. In the next section we will see that the spotlight principle is inherently capable of comprising all instantiations of a fully symmetric system in one abstract model, and thus, definite results obtained on such a model can be transferred to the corresponding parameterised system.

6.3 Symmetry and Spotlight

So far, we have shown that symmetry enables us to reduce checking global requirements to checking local requirements – which works for fixed instances of parameterised systems. Thus, we still have to bridge the gap between verifying *single* instantiations and *complete* parameterised verification. With this in mind, we look again at our running example: the instantiation Sys_4 of the fully symmetric system Sys from Figure 6.1 and the local requirement $\mathbf{AG}\neg(pc_1 = 4 \wedge pc_2 = 4)$. The spotlight abstraction of Sys_4 depicted in Figure 6.3 is already precise enough to validate the mutual exclusion property.

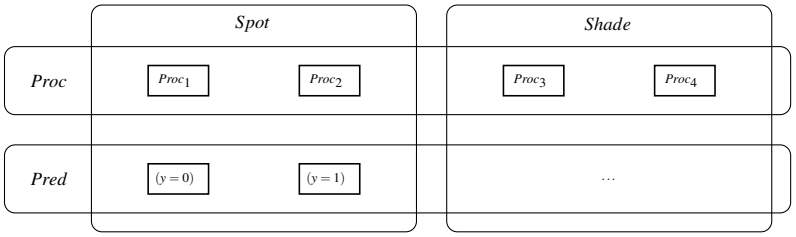

Fig. 6.3 Spotlight abstraction of the instantiation Sys_4 (compare Figure 6.2) with $Spot(Proc) = \{Proc_1, Proc_2\}$ and $Spot(Pred) = \{(y = 0), (y = 1)\}$.

Hence, we only have to consider the processes $Proc_1$ and $Proc_2$ in detail (which we already mentioned in the section before), and moreover, we require two atomic predicates over the shared semaphore variable y. Model checking the formula $\mathbf{AG}\neg(pc_1 = 4 \wedge pc_2 = 4)$ on a Kripke structure corresponding to this spotlight abstraction yields *true*, and based on symmetry arguments we can conclude that the global property $\bigwedge_{\langle i_1, i_2\rangle \in [PID_4]^2_{\neq}} \mathbf{AG}\neg(pc_{i_1} = 4 \wedge pc_{i_2} = 4)$ holds as well for the instantiation Sys_4.

For the moment, we disregard the verification result, and instead, take a closer look at the nature of this abstraction. The spotlight principle has been thoroughly introduced in Chapter 4. One of its key features is the summarisation of the shade processes into *one*[1] *component*: an abstract process $Proc_{Shade}$ that continuously executes one operation that approximates all concrete operations on shared variables occurring in the shade (compare Definition 4.8). In our running example the shade consists of two replicated processes that both modify the shared semaphore y by the operations $acquire(y, 1)$ and $release(y, 1)$. An appropriate shade component for $Shade(Proc) = \{Proc_3, Proc_4\}$ and $Spot(Pred) = \{(y = 0), (y = 1)\}$ is given in Figure 6.4.

The abstract operation bop_{Shade} approximates $acquire(y, 1)$ as well as $release(y, 1)$. Since these two semaphore operations are the only modifications of shared variables in $Proc_3$ and $Proc_4$, we have that $Proc_{Shade}$ is an

[1] We assume the basic variant of spotlight abstraction.

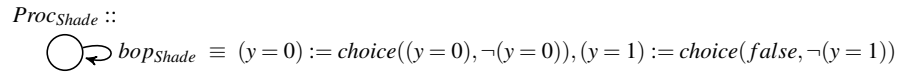

Fig. 6.4 Control flow representation of the shade component $Proc_{Shade}$ corresponding to the spotlight abstraction of Sys_4.

admissible shade component for our current abstraction. Apparently, multiple occurrences of the same operation in the shade do not require any special treatment. If bop_{Shade} approximates an operation bop from some shade process, then any further occurrence of bop in the shade is of course also approximated by bop_{Shade}. Exactly this fact can be exploited for our approach to parameterised verification: A fully symmetric system consists of replicated processes that all execute *identical* operations. Thus, *any* number of these replications in the shade will give us the same shade component. This allows us to transfer verification results obtained on a spotlight abstraction of a single instance to any larger instantiation:

Theorem 6.2.
Let $Sys = \|_{i \in PID_N} Proc_i$ be a fully symmetric parameterised system and let $Sys_n = \|_{i \in PID_n} Proc_i$ be a fixed instantiation of Sys. Moreover, let $Spot = Spot(Proc) \cup Spot(Pred)$, $Shade = Shade(Proc) \cup Shade(Pred)$ be a given spotlight abstraction for Sys_n with $Shade(Proc) \neq \emptyset$. Let $Sys_n^a = \|_{Proc_i \in Spot(Proc)} Proc_i^a \| Proc_{Shade}$ be the corresponding abstract system with $Sys_n^a \preceq Sys_n$, $K(Sys_n^a) = (S^a, R^a, L^a, \mathbb{F}^a)$ the respective abstract Kripke structure over $AP = Spot(Pred) \cup \{pc_i = j | Proc_i \in Spot(Proc), j \in Loc_i\}$, and $\psi(pid_1, \ldots, pid_d)$ a CTL formula over AP where $pid_1, \ldots, pid_d$ are pairwise different identifiers of processes in $Spot(Proc)$. Then for any instantiation $Sys_m = \|_{i \in PID_m} Proc_i$ of Sys with $m \geq n$, $PID_m = PID_n \cup \{pid_{n+1}, \ldots, pid_m\}$ and the respective concrete Kripke structure $K(Sys_m) = (S, R, L, \mathbb{F})$ over AP, and for any pair of corresponding[2] states $s^a \in S^a$, $s \in S$:

$$[K(Sys_n^a), s^a \models \psi(pid_1, \ldots, pid_d)] \leq_{\mathbb{K}_3} [K(Sys_m), s \models \psi(pid_1, \ldots, pid_d)]$$

Proof (Theorem 6.2).
We have given an instantiation $Sys_n = \|_{i \in PID_n} Proc_i$ of a fully symmetric system Sys, and a spotlight abstraction $Spot = Spot(Proc) \cup Spot(Pred)$, $Shade = Shade(Proc) \cup Shade(Pred)$ for Sys_n with $Shade(Proc) \neq \emptyset$. According to Definition 3.3 all processes in fully symmetric systems are identical, i.e. they are replications of each other. Hence, we can extend the given instantiation to the next larger one as follows: $Sys_{n+1} := Sys_n \| Proc_j$ where $Proc_i$ is a replication of an arbitrary process of $Shade(Proc)$. Moreover, we extend the spotlight abstraction for the enlarged instantiation: $Spot := Spot$ and $Shade := Shade \cup \{Proc_j\}$, i.e. we put the additional process into the shade,

[2] Compare Definition 4.7.

whereas the spotlight is not affected. From Definition 4.8 (spotlight abstraction of concurrent systems) we can deduce that the shade component $Proc_{Shade}$ remains the same when we add a replication of a process $Proc_i \in Shade(Proc)$ to the shade. Thus, the abstract systems Sys^a_n, Sys^a_{n+1} corresponding to Sys_n resp. Sys_{n+1} are identical. Corollary 4.3 allows us to transfer all definite verification results obtained for the abstract system Sys^a_n to the corresponding original system Sys_n, and due to identity of Sys^a_n and Sys^a_{n+1} these results can be further transferred to Sys_{n+1}. By induction we get the same results for all instantiations Sys_m with $m > n$.
□

Hence, if we construct a spotlight abstraction of a fixed instantiation of a fully symmetric system and checking a temporal logic property yields *true* or *false*, then we can conclude that this outcome holds for all larger instantiations as well. However, there still remain some limitations: First, the shade has to contain at least one process – otherwise our spotlight abstraction would not reflect the behaviour of an arbitrary number of processes. And second, the checked property has to be a local one that solely refers to the finite set of processes in the spotlight. Nevertheless, the latter limitation can be straightforwardly resolved based on our symmetry arguments. Corollary 6.2 together with Theorem 6.2 gives us the following result:

Corollary 6.3.
Let Sys, $K(Sys^a_n)$, $K(Sys_m)$ (for all $m \geq n$), and $\psi(pid_1, \ldots, pid_d)$ be defined as in Theorem 6.2. Then for all $m \geq n$, and for any pair of corresponding states $s^a \in S^a$, $s \in S$:

$$[K(Sys^a_n), s^a \models \psi(pid_1, \ldots, pid_d)]$$
$$\leq_{\mathbb{K}_3}$$
$$[K(Sys_m), s \models \bigwedge_{\langle i_1,\ldots,i_d\rangle \in [PID_m]^d_{\neq}} \psi(i_1, \ldots, i_d)]$$

According to this corollary, we can perform parameterised verification as follows: As an input, we have a fully symmetric system $Sys = \|_{i \in PID_N} Proc_i$ and a global CTL formula $\Psi = \bigwedge_{\langle i_1,\ldots,i_d\rangle \in [PID_m]^d_{\neq}} \psi(i_1, \ldots, i_d)$. In the first step, we create a finite instantiation $Sys_n = \|_{i \in PID_n} Proc_i$ of Sys with $n > d$, i.e. we require at least one more process in our instantiation than the number of different process identifiers in Ψ. Second, we select one clause $\psi(pid_1, \ldots, pid_d)$ of Ψ with $pid_1, \ldots, pid_d \in PID_n$, which means $\psi(pid_1, \ldots, pid_d)$ is a local property with regard to Sys_n. Hence, checking whether $\psi(pid_1, \ldots, pid_d)$ holds for Sys_n can be straightforwardly done with our spotlight abstraction refinement framework. Now symmetry allows us to transfer the obtained result to the corresponding global verification task, i.e. whether Ψ holds for Sys_n. And finally, based on the nature of spotlight abstraction, this result can be further transferred to any instantiation larger than Sys_n – which implies that we have successfully verified the parameterised system Sys.

However, since parameterised verification is undecidable in general there is still a catch: Checking temporal logic properties on three-valued spotlight abstractions may also return *unknown*, which tells us nothing about the parameterised system. And moreover, our final conclusion step is only permitted in the case of a non-empty shade. Thus, any definite result obtained on an abstraction where all processes are in the spotlight has to treated as *unknown* as well. So our approach gives us a sound but incomplete procedure for parameterised verification.

So far, we have only considered fully symmetric systems where *all* processes are homogeneous. Indeed, not many real-life systems fit into this category. In the next section, we will see that our approach can be extended to *class-wise symmetric systems* where we can distinguish individual classes of homogeneous processes. This extension enables us to cover more systems of practical relevance in our approach, and moreover, it forms the basis for integrating parameterised verification into our heuristic framework for abstraction refinement.

6.4 Relaxed Symmetry

In Chapter 3 we introduced class-wise symmetric systems. Such parameterised systems are not fully symmetric, but they consist of a finite number of *classes* of fully symmetric processes. Algorithms for the *producers-consumers problem*, the *sleeping barber problem* and the *readers-writers problem* are classical examples of systems where we have different classes of homogeneous processes, and generally an arbitrary number of processes per class. In fact, diverse network protocols are based on slight extensions of these algorithms. Thus, allowing for *class-wise symmetry* is a relaxation on the structure of a parameterised system that will permit us to consider a much larger variety of concurrent computation algorithms in our verification framework. Subsequently, we show how this relaxation can be soundly established for our approach. We first take a look at a simple example of a class-wise symmetric system. In Section 3.2 we already discussed the following solution for the readers-writers problem:

$$y : \textbf{semaphore where } y = N_{Rd}$$

$$\|_{i \in PID_{N_{Rd}}^{Rd}} Rd_i :: \left[\begin{array}{ll} 1: & \textbf{loop forever do} \\ & \left[\begin{array}{ll} 2: & \textbf{non-critical} \\ 3: & \textbf{acquire}(y,1) \\ 4: & \textbf{critical-read} \\ 5: & \textbf{release}(y,1) \end{array}\right] \end{array}\right] \|_{j \in PID_{N_{Wrt}}^{Wrt}} Wrt_j :: \left[\begin{array}{ll} 1: & \textbf{loop forever do} \\ & \left[\begin{array}{ll} 2: & \textbf{non-critical} \\ 3: & \textbf{acquire}(y,N_{Rd}) \\ 4: & \textbf{critical-write} \\ 5: & \textbf{release}(y,N_{Rd}) \end{array}\right] \end{array}\right]$$

Fig. 6.5 Class-wise symmetric system $Sys = \|_{i \in PID_{N_{Rd}}^{Rd}} Rd_i \|_{j \in PID_{N_{Wrt}}^{Wrt}} Wrt_j$ consisting of a reader class Rd and a writer class Wrt. $PID_{N_{Rd}}^{Rd}$ and $PID_{N_{Wrt}}^{Wrt}$ are sets of process indices with parameterised sizes $N_{Rd} \in \mathbb{N}$ resp. $N_{Wrt} \in \mathbb{N}$.

Here we have two classes of processes: the reader class *Rd* and the writer class *Wrt*. Reader processes as well as writer processes continuously attempt to enter a critical section which is protected by one semaphore y. The semaphore has a parameterised capacity of N_{Rd}, which corresponds to the actual number of readers in the system. As we can see, readers have to acquire a single unit of the semaphores capacity, while writers require the full capacity of y. Hence, an arbitrary number of readers can enter the critical section simultaneously, whereas writers have mutually exclusive access to the critical section. This observed fact about *Sys* corresponds to the general correctness requirement for readers-writers systems: *Never a reader and a writer in the critical section at the same time, and never two writers in the critical section at the same time.* This can be formalised in temporal logic as follows:

$$\Psi := \bigwedge_{\langle i_1^{Rd}, i_1^{Wrt}, i_2^{Wrt} \rangle \in PID_{N_{Rd}}^{Rd} \times \left[PID_{N_{Wrt}}^{Wrt}\right]_{\neq}^{2}} \left(\mathbf{AG}\neg\left(pc_{i_1^{Rd}} = 4 \wedge pc_{i_1^{Wrt}} = 4\right) \wedge \mathbf{AG}\neg\left(pc_{i_1^{Wrt}} = 4 \wedge pc_{i_2^{Wrt}} = 4\right)\right)$$

with

$$\left[PID_{N_{Wrt}}^{Wrt}\right]_{\neq}^{2} := \left\{\langle i, i' \rangle \mid i, i' \in PID_{N_{Wrt}}^{Wrt},\ i \neq i'\right\},$$

i.e. $[PID_{N_{Wrt}}^{Wrt}]_{\neq}^{2}$ denotes the second cartesian power of the set $PID_{N_{Wrt}}^{Wrt}$ where all tuples $\langle i, i' \rangle \in [PID_{N_{Wrt}}^{Wrt}]_{\neq}^{2}$ consist of *pairwise different* process identifiers from $PID_{N_{Wrt}}^{Wrt}$.

The abovementioned correctness requirement refers to *one* arbitrary reader and to *two* arbitrary but pairwise different writers. Hence, the corresponding global temporal logic formula Ψ is a conjunction over all possible 3-tuples $\langle i_1^{Rd}, i_1^{Wrt}, i_2^{Wrt} \rangle$ of pairwise different process identifiers with the appropriate class affiliation. A parameterised system satisfies such a global property if and only if it holds for any possible instantiation. In contrast to fully symmetric systems the size of an instantiation of a class-wise symmetric system is not characterised by a single parameter. For each class a specific number of processes has to be instantiated. Given a class-wise symmetric system *Sys* with k classes, we denote an instantiation by $Sys_{n_1,\ldots,n_k}$ where $n_1, \ldots, n_k \in \mathbb{N}$ are the respective numbers of processes of each class. An instantiation $Sys_{2,2}$ of our readers-writers system is depicted in Figure 6.6.

In fact, the previously specified global CTL formula Ψ holds for this particular instantiation of *Sys*. However, we want to verify whether we get the same result for *all* instantiations:

$$\forall n_{Rd} \geq 2\ \forall n_{Wrt} \geq 2\ :\ K\left(Sys_{n_{Rd}, n_{Wrt}}\right), s_0 \ \models\ \Psi$$

$$y : \textbf{semaphore where } y = 2$$

$$Rd_1 :: \left[\begin{array}{ll} 1: & \textbf{loop forever do} \\ & \left[\begin{array}{ll} 2: & \textbf{non-critical} \\ 3: & \textbf{acquire}(y,1) \\ 4: & \textbf{critical-read} \\ 5: & \textbf{release}(y,1) \end{array}\right] \end{array}\right] \parallel Rd_2 :: \left[\begin{array}{ll} 1: & \textbf{loop forever do} \\ & \left[\begin{array}{ll} 2: & \textbf{non-critical} \\ 3: & \textbf{acquire}(y,1) \\ 4: & \textbf{critical-read} \\ 5: & \textbf{release}(y,1) \end{array}\right] \end{array}\right]$$

$$\parallel Wrt_a :: \left[\begin{array}{ll} 1: & \textbf{loop forever do} \\ & \left[\begin{array}{ll} 2: & \textbf{non-critical} \\ 3: & \textbf{acquire}(y,2) \\ 4: & \textbf{critical-write} \\ 5: & \textbf{release}(y,2) \end{array}\right] \end{array}\right] \parallel Wrt_b :: \left[\begin{array}{ll} 1: & \textbf{loop forever do} \\ & \left[\begin{array}{ll} 2: & \textbf{non-critical} \\ 3: & \textbf{acquire}(y,2) \\ 4: & \textbf{critical-write} \\ 5: & \textbf{release}(y,2) \end{array}\right] \end{array}\right]$$

Fig. 6.6 Instantiation $Sys_{2,2}$ of the class-wise symmetric system Sys with $PID_2^{Rd} = \{1,2\}$ and $PID_2^{Wrt} = \{a,b\}$.

where $K(Sys_{n_{Rd},n_{Wrt}})$ is a Kripke structure corresponding to the instantiation $Sys_{n_{Rd},n_{Wrt}}$ and s_0 is the state of $K(Sys_{n_{Rd},n_{Wrt}})$ that represents the initial configuration of $Sys_{n_{Rd},n_{Wrt}}$. This model checking task exemplifies parameterised verification for our simple readers-writers system. The general definitions of global CTL formulae and parameterised verification of class-wise symmetric systems are given below.

Definition 6.6 (Global CTL Formulae over Class-Wise Sym. Systems). Let $Sys = \|_{m=1}^{k} (\|_{i \in PID_{N_m}^m} Proc_i^m)$ be a class-wise symmetric system. Moreover, let $\psi(i_1^1,\ldots,i_{d_1}^1,\ldots,i_1^k,\ldots,i_{d_k}^k)$ is a parameterised CTL formula with reference to variables for process identifiers from $PID_{N_1}^1$ to $PID_{N_k}^k$, respectively. Then the corresponding *global CTL formula* is

$$\Psi := \bigwedge_{\langle i_1^1,\ldots,i_{d_1}^1,\ldots,i_1^k,\ldots,i_{d_k}^k\rangle \in \times_{m=1}^{k}([PID_{N_m}^m]_{\neq}^{d_m})} \psi(i_1^1,\ldots,i_{d_1}^1,\ldots,i_1^k,\ldots,i_{d_k}^k).$$

Thus, for each $1 \leq m \leq k$: $[PID_{N_m}^m]_{\neq}^{d_m}$ denotes the d_m-th cartesian power of the set $PID_{N_m}^m$ where all d_m-tuples $\langle i_1^m,\ldots,i_{d_m}^m\rangle \in [PID_{n_m}^m]_{\neq}^{d_m}$ consist of pairwise different process identifiers from $PID_{n_m}^m$. Consequently, the global formula Ψ is a conjunction over all possible combinations of $d_1,\ldots d_k$ pairwise different processes from each class $1,\ldots,k$, where each clause of Ψ corresponds to a local CTL formula that refers to $d_1,\ldots,d_k$ specific processes per class $1,\ldots,k$.

Definition 6.7 (Parameterised Verification of Class-Wise Sym. Systems). Let $Sys = \|_{m=1}^{k} (\|_{i \in PID_{N_m}^m} Proc_i^m)$ be a class-wise symmetric system and let $\Psi = \bigwedge_{\langle i_1^1,\ldots,i_{d_1}^1,\ldots,i_1^k,\ldots,i_{d_k}^k\rangle \in \times_{m=1}^{k}([PID_{N_m}^m]_{\neq}^{d_m})} \psi(i_1^1,\ldots,i_{d_1}^1,\ldots,i_1^k,\ldots,i_{d_k}^k)$ be a global CTL formula over Sys. Then the corresponding *parameterised verification problem* is

$$\forall n_1 \geq d_1 \ \ldots \ \forall n_k \geq d_k \ :$$

$$K\left(Sys_{n_1,\ldots,n_k}\right), s_0$$
$$\models$$
$$\bigwedge_{\left\langle i^1_1,\ldots,i^1_{d_1},\ldots,i^k_1,\ldots,i^k_{d_k}\right\rangle \in \times_{m=1}^{k}\left(\left[PID^m_{n_m}\right]^{d_m}_{\neq}\right)} \psi\left(i^1_1,\ldots,i^1_{d_1},\ldots,i^k_1,\ldots,i^k_{d_k}\right)$$

where $K(Sys_{n_1,\ldots,n_k})$ is a Kripke structure corresponding to an instantiation $Sys_{n_1,\ldots,n_k}$ of Sys and s_0 is the state of $K(Sys_{n_1,\ldots,n_k})$ that represents the initial configuration of $Sys_{n_1,\ldots,n_k}$.

Subsequently, we will show that our approach to parameterised verification based on symmetry reduction and spotlight abstraction can be easily transferred to class-wise symmetric systems. Here we have to deal with a relaxed notion of symmetry, and thus, we also require a new notion of process permutations:

Definition 6.8 (Class-Sensitive Process Permutation).
Let $Sys_{n_1,\ldots,n_k} = \|^k_{m=1}\left(\|_{i\in PID^m_{n_m}} Proc^m_i\right)$ be an instantiation of a class-wise symmetric parameterised system Sys. Then a corresponding *class-sensitive process permutation* is a bijective function

$$\sigma : \bigcup_{m=1}^{k} PID^m_{n_m} \to \bigcup_{m=1}^{k} PID^m_{n_m}.$$

with $i^m \in PID^m_{n_m} \Leftrightarrow \sigma(i^m) \in PID^m_{n_m}$ for all $1 \leq m \leq k$ and all $i^m \in PID^m_{n_m}$.

As we can see, a class-sensitive permutation σ preserves the class affiliation of process identifiers. We can even *decompose* σ into k unrestricted process permutations $\sigma^m : PID^m_{n_m} \to PID^m_{n_m}$ with $1 \leq m \leq k$. On the other hand, a class-wise symmetric system Sys can be regarded as a *composition* of k fully symmetric systems, i.e. $Sys = \|^k_{m=1} Sys^m$ where $Sys^m = \|_{i\in PID^m_{n_m}} Proc^m_i$. Hence, each partial permutation σ^m is a *symmetry* for the subsystem Sys^m, and by induction we get the following result:

Lemma 6.2.
On Kripke structures corresponding to instantiations of class-wise symmetric systems, all class-sensitive process permutations are symmetries.

In the next step, we consider the preservation of CTL properties of class-wise symmetric systems under class-sensitive permutations. We regard properties of the form $\Psi = \bigwedge_{\langle i^1_1,\ldots,i^1_{d_1},\ldots,i^k_1,\ldots,i^k_{d_k}\rangle \in \times_{m=1}^{k}([PID^m_{n_m}]^{d_m}_{\neq})} \psi(i^1_1,\ldots,i^1_{d_1},\ldots,i^k_1,\ldots,i^k_{d_k})$.
Definition 6.6 allows us rewrite such global CTL formulae as follows:

$$\Psi \;\Leftrightarrow\; \bigwedge_{\sigma\in\Sigma} \psi(\sigma(pid^1_1),\ldots,\sigma(pid^1_{d_1}),\ldots,\sigma(pid^k_1),\ldots,\sigma(pid^k_{d_k}))$$

where Σ is the set of all class-sensitive process permutations over $\bigcup_{m=1}^{k} PID_{n_m}^m$, and for all $1 \leq m \leq k$: $pid_1^m, \ldots, pid_{d_m}^m$ are arbitrary, pairwise different process identifiers from $PID_{n_m}^m$. Now analogous to Theorem 6.1 and Corollary 6.2, we obtain the following result for class-wise symmetric systems:

Corollary 6.4.
Let $Sys_{n_1,\ldots,n_k} = \|_{m=1}^{k} (\|_{i \in PID_{n_m}^m} Proc_i^m)$ be an instantiation of a class-wise symmetric system, $K = (S, R, L, \mathbb{F})$ be the corresponding Kripke structure over a set of atomic predicates AP, and Σ the set of all class-sensitive process permutations for Sys. Moreover, let $\psi(pid_1^1, \ldots, pid_{d_1}^1, \ldots, pid_1^k, \ldots, pid_{d_k}^k)$ be a CTL formula over AP with $\langle pid_1^1, \ldots, pid_{d_1}^1, \ldots, pid_1^k, \ldots, pid_{d_k}^k \rangle \in \bigtimes_{m=1}^{k} ([PID_{n_m}^m]_{\neq}^{d_m})$ and let $s_0 \in S$ be the state of K that represents the initial configuration of $Sys_{n_1,\ldots,n_k}$. Then

$$[K, s_0 \models \psi(pid_1^1, \ldots, pid_{d_1}^1, \ldots, pid_1^k, \ldots, pid_{d_k}^k)]$$

$$\Leftrightarrow$$

$$[K, s_0 \models \bigwedge_{\sigma \in \Sigma} \psi(\sigma(pid_1^1), \ldots, \sigma(pid_{d_1}^1), \ldots, \sigma(pid_1^k), \ldots, \sigma(pid_{d_k}^k))].$$

Thus, applying class-sensitive process permutations preserves the validity of local CTL properties that refer to particular processes of a class-wise symmetric system. And again, this allows us to reduce global verification tasks to local ones.

In the final step, we show how parameterised verification of class-wise symmetric systems can be combined with spotlight abstraction. The general approach is the same as for fully symmetric systems. We have only one additional requirement: The shade has to contain at least one process of *each* class. Hence, the shade component summarises the behaviour of an arbitrary number of processes from all classes. Analogous to Corollary 6.3 we get:

Corollary 6.5.
Let $Sys = \|_{m=1}^{k} (\|_{i \in PID_{N_m}^m} Proc_i^m)$ be a class-wise symmetric parameterised system and let $Sys_{n_1,\ldots,n_k} = \|_{m=1}^{k} (\|_{i \in PID_{n_m}^m} Proc_i^m)$ be a fixed instantiation of Sys. Moreover, let $Spot = Spot(Proc) \cup Spot(Pred)$, $Shade = Shade(Proc) \cup Shade(Pred)$ be a given spotlight abstraction for $Sys_{n_1,\ldots,n_k}$ with $\forall 1 \leq m \leq k\ \exists i \in PID_{n_m}^m$: $Proc_i^m \in Shade(Proc)$. Let $Sys_{n_1,\ldots,n_k}^a = \|_{Proc_i^m \in Spot(Proc)} Proc_i^{m^a} \parallel Proc_{Shade}$ be the corresponding abstract system with $Sys_{n_1,\ldots,n_k}^a \preceq Sys_{n_1,\ldots,n_k}$, $K(Sys_{n_1,\ldots,n_k}^a) = (S^a, R^a, L^a, \mathbb{F}^a)$ the respective abstract Kripke structure over $AP = Spot(Pred) \cup \{pc_i = j \mid Proc_i^m \in Spot(Proc), j \in Loc_i\}$, and $\psi(pid_1^1, \ldots, pid_{d_1}^1, \ldots, pid_1^k, \ldots, pid_{d_k}^k)$ a temporal logic formula over AP where $\langle pid_1^1, \ldots, pid_{d_1}^1, \ldots, pid_1^k, \ldots, pid_{d_k}^k \rangle \in \bigtimes_{m=1}^{k} ([PID_{n_m}^m]_{\neq}^{d_m})$ is a tuple of class-affiliated, pairwise different identifiers of processes in $Spot(Proc)$. Then for any instantiation $Sys_{n'_1,\ldots,n'_k} = \|_{m=1}^{k} (\|_{i \in PID_{n'_m}^m} Proc_i^m)$ of Sys with $\forall 1 \leq m \leq k : n'_m \geq n_m$, $PID_{n'_m}^m = PID_{n_m}^m \cup \{pid_{n+1}^m, \ldots, pid_{n'}^m\}$ and the respective concrete Kripke structure $K(Sys_{n'_1,\ldots,n'_k}) = (S, R, L, \mathbb{F})$ over AP, and for any pair of corresponding states $s^a \in S^a$, $s \in S$:

$$K\left(Sys^{a}_{n_1,\ldots,n_k}\right), s^a \models \psi\left(pid^1_1,\ldots,pid^1_{d_1},\ldots,pid^k_1,\ldots,pid^k_{d_k}\right)$$

$$\leq_{\mathbb{K}_3}$$

$$K\left(Sys_{n'_1,\ldots,n'_k}\right), s \models \bigwedge_{\left\langle i^1_1,\ldots,i^1_{d_1},\ldots,i^k_1,\ldots,i^k_{d_k}\right\rangle \in \times_{m=1}^{k}\left(\left[PID^m_{n_m}\right]^{d_m}_{\neq}\right)} \psi\left(i^1_1,\ldots,i^1_{d_1},\ldots,i^k_1,\ldots,i^k_{d_k}\right)$$

In order to illustrate how this result can be utilised for parameterised verification we consider again our running example. We want to validate whether the readers-writers system *Sys* in Figure 6.5 is correct, i.e. whether there will never be a reader and a writer in the critical section at the same time, and never two writers in the critical section at the same time. This global requirement refers to one arbitrary reader and two arbitrary but distinct writers. For each distinct process referenced in the requirement we need one representative process in the spotlight. Moreover, for approximating the behaviour of an arbitrary number of processes we additionally need one process of each class inside the shade. Therefore, we construct an instantiation $Sys_{2,3} = Rd_1 \parallel Rd_2 \parallel Wrt_a \parallel Wrt_b \parallel Wrt_c$ of *Sys*, and we partition the systems processes into spotlight and shade as follows: $Spot(Proc) = \{Rd_1, Wrt_a, Wrt_b\}$ and $Shade(Proc) = \{Rd_2, Wrt_c\}$. In the next step, we narrow down the global requirement to a local CTL property that just refers our representative spotlight processes: $\mathbf{AG}\neg(pc_1 = 4 \wedge pc_a = 4) \wedge \mathbf{AG}\neg(pc_a = 4 \wedge pc_b = 4)$. Model checking this formula on a Kripke structure corresponding to the current spotlight abstraction yields *true*. According to Corollary 6.5 we get the same result for any class-sensitive permutation of the local property, and consequently, also for our global requirement. Moreover, this result can be further transferred to *any* instantiation of *Sys* that is larger than $Sys_{2,3}$. Hence, symmetry reduction combined with the spotlight principle, allows us to successfully verify a class-wise symmetric parameterised system on a very small abstraction.

Our approach to the verification of parameterised systems generally requires the selection of a preferably small instantiation that is already large enough for a definite result. A basic heuristic is to select one representative spotlight process for each distinct process referenced in the property to be checked, and additionally, one shade process for each class of the parameterised system. By our running example we have demonstrated the effectiveness of this heuristic for an individual case, and in fact it also works well for several other verification tasks. However, many properties of class-wise symmetric systems arise from a *complex interplay* between a cluster of processes, which cannot be captured by an instantiation according to our basic heuristic. For instance, the liveness formula $\bigwedge_{\langle i^{Rd}\rangle \in PID^{Rd}_{N_{Rd}}} \mathbf{AG}((pc_{i^{Rd}} = 3) \Rightarrow \mathbf{AF}(pc_{i^{Rd}} = 4))$ refers to a *single* process of the readers-writers system *Sys*. But its verification requires at least one *additional* process to be in the spotlight that competes for the same semaphore. An essentially more complex example would be the verification of a parameterised producer-consumer system with multiple resources. Here the liveness of a con-

sumer process depends on the behaviour of potential partners (producers) and competitors (other consumers). Moreover, a producer of a certain resource might be in turn a consumer of another resource, and so forth. Hence, chains and cycles of dependencies may necessitate nontrivial clusters of processes to be inside the spotlight in order to obtain a definite verification result. – In the next section we show that, again, heuristic guidance can help us to iteratively construct appropriate instantiations (and abstractions) of class-wise symmetric systems in parameterised verification.

6.5 Abstraction Refinement

Symmetry in parameterised systems does not necessarily come along with pure uniformity. The more symmetry classes we have in a system, the more non-uniform dependencies we get in potential instantiations – which in turn makes the task of finding the right degree of abstraction more challenging. In this section we show how our framework for heuristic-guided abstraction refinement can be adopted for the iterative construction of abstractions of *class-wise symmetric systems*. The first issue that we encounter here is that parameterised verification does not only require the selection of an adequate abstraction, but also the selection of an adequately sized *instantiation*. Subsequently, we will see that our framework is capable of accomplishing both selections in one step. Based on the parameterised message passing system Sys_{MP} depicted in Figure 6.7 we illustrate our approach.

$$c,d,e,f,g : \textbf{channel}\ [1]\ \textbf{of integer}$$

$$\|_{i\in PID^1_{N_1}}\ Proc^1_i :: \begin{bmatrix} 1: & \textbf{loop forever do} \\ & \begin{bmatrix} 2: & \textbf{receive}(c,\ \cdot\ ) \\ 3: & \textbf{receive}(d,\ \cdot\ ) \\ 4: & \textbf{progress} \end{bmatrix} \end{bmatrix} \quad \|_{i\in PID^2_{N_2}}\ Proc^2_i :: \begin{bmatrix} 1: & \textbf{loop forever do} \\ & \begin{bmatrix} 2: & \textbf{send}(d,\ \cdot\ ) \\ 3: & \textbf{progress} \end{bmatrix} \end{bmatrix}$$

$$\|_{i\in PID^3_{N_3}}\ Proc^3_i :: \begin{bmatrix} 1: & \textbf{loop forever do} \\ & \begin{bmatrix} 2: & \textbf{receive}(e,\ \cdot\ ) \\ 3: & \textbf{send}(c,\ \cdot\ ) \\ 4: & \textbf{progress} \end{bmatrix} \end{bmatrix} \quad \|_{i\in PID^4_{N_4}}\ Proc^4_i :: \begin{bmatrix} 1: & \textbf{loop forever do} \\ & \begin{bmatrix} 2: & \textbf{receive}(d,\ \cdot\ ) \\ 3: & \textbf{progress} \end{bmatrix} \end{bmatrix}$$

$$\|_{i\in PID^5_{N_5}}\ Proc^5_i :: \begin{bmatrix} 1: & \textbf{loop forever do} \\ & \begin{bmatrix} 2: & \textbf{receive}(f,\ \cdot\ ) \\ 3: & \textbf{send}(e,\ \cdot\ ) \\ 4: & \textbf{progress} \end{bmatrix} \end{bmatrix} \quad \|_{i\in PID^6_{N_6}}\ Proc^6_i :: \begin{bmatrix} 1: & \textbf{loop forever do} \\ & \begin{bmatrix} 2: & \textbf{receive}(g,\ \cdot\ ) \\ 3: & \textbf{send}(f,\ \cdot\ ) \\ 4: & \textbf{progress} \end{bmatrix} \end{bmatrix}$$

Fig. 6.7 Class-wise symmetric message passing system $Sys_{MP} = \|^6_{m=1} \left(\|_{i\in PID^m_{N_m}}\ Proc^m_i \right)$ over $Var = \{c,d,e,f,g\}$. For simplification, the data values of communication statements are omitted.

As we can see, the communication structure of this system exhibits a high degree of non-uniform dependencies, which makes verification generally difficult. Before we consider a concrete verification task for Sys_{MP} we show how an initial abstraction and a corresponding dependence graph can be constructed for an arbitrary class-wise symmetric system $Sys = \|_{m=1}^{k} (\|_{i \in PID_{N_m}^m} Proc_i^m)$. The set of different processes of Sys is $\{Proc^1, \ldots, Proc^k\}$, i.e. each of these processes represents a distinct class. We now initialise a spotlight abstraction of Sys as follows: $Spot = \emptyset$ and $Shade(Proc) = \{Proc^1, \ldots, Proc^k\}$. Moreover, we assume that the initial set of refinement candidates is empty, i.e. $Candidates = \emptyset$. According to Definition 5.5 we can construct a corresponding abstraction dependence graph $ADG = (V, D)$. Contrary to ADGs of fixed instantiations, we now have that each node $v \in V$ that is associated with $Shade(Proc)$ represents a class of an *arbitrary* number of processes. Since distinct processes of the same class might affect each other, we extend the dependence relation of the abstraction dependence graph $ADG = (V, D)$ as follows:

$$D := D \cup \{(v,v) \mid ((v \in Proc(Shade) \wedge \exists\, x (x \in DEF(v) \wedge x \in REF(v))))\},$$

i.e. the dependencies can now be *reflexive* for vertices in $Proc(Shade)$.

Hence, we start with an entirely empty spotlight and for each class of processes we introduce one vertex inside the shade. Such a vertex now represents an arbitrary number of processes of the same kind. Thus, we may not only have dependencies between processes of different classes, but also between processes *within* a class. In Figure 6.8 we see the initial abstraction dependence graph corresponding to the message passing system Sys_{MP}.

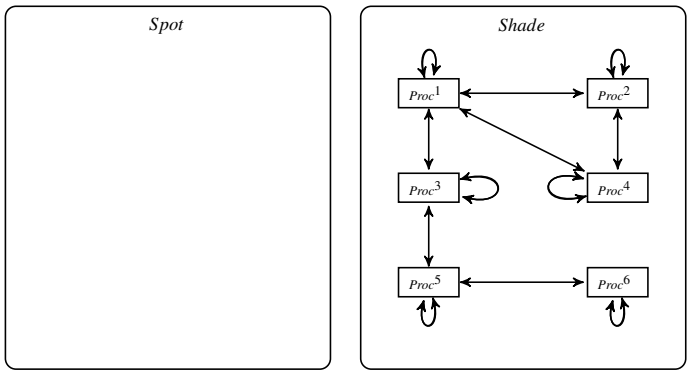

Fig. 6.8 Abstraction dependence graph corresponding to the spotlight abstraction $Spot = \emptyset$ and $Shade(Proc) = \{Proc^1, Proc^2, Proc^3, Proc^4, Proc^5, Proc^6\}$ of the class-wise symmetric message passing system $Sys_{MP} = \|_{m=1}^{6} (\|_{i \in PID_{N_m}^m} Proc_i^m)$.

We now want to check whether all processes of class 1 of the parameterised system will continuously reach their critical section, i.e. whether the liveness

formula

$$\Psi := \bigwedge_{\langle i \rangle \in PID^1_{N_1}} \mathbf{AG}\,((pc_i = 3) \;\Rightarrow\; \mathbf{AF}(pc_i = 4))$$

holds for all possible instantiations of Sys_{MP}. The global CTL formula Ψ refers to one arbitrary process of class 1. Hence, we create a new instantiation of $Proc^1$ inside the spotlight – *without* removing $Proc^1$ from the shade. The instantiation $Proc^1_1$ inherits all but the reflexive dependencies of the shade vertex $Proc^1$. The corresponding abstraction dependence graph is shown in Figure 6.9. The inherited dependencies are depicted as dashed edges.

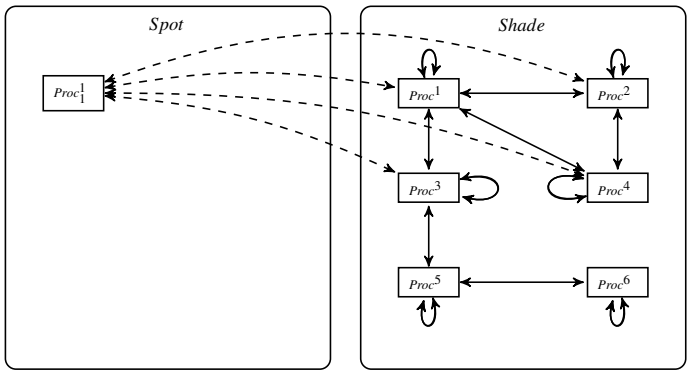

Fig. 6.9 Abstraction dependence graph corresponding to the spotlight abstraction $Spot = \{Proc^1_1\}$ and $Shade(Proc) = \{Proc^1, Proc^2, Proc^3, Proc^4, Proc^5, Proc^6\}$ of the class-wise symmetric message passing system $Sys_{MP} = \|^6_{m=1}\,(\|_{i \in PID^m_{N_m}}\, Proc^m_i)$.

This nearly completes our extended abstraction refinement framework for class-wise symmetric systems. We have determined the initial spotlight for our current verification task and we have constructed the corresponding abstraction dependence graph. Now, the steps *model checking*, *counterexample generation* and *heuristic selection of the most promising refinement candidate* can be iteratively performed as introduced in Chapter 5 – with the slight difference that processes are not shifted from the shade to the spotlight, but *new instantiations* of processes are created inside the spotlight. Since process vertices are never removed from the shade, this procedure might not terminate, which comes along with the fact that parameterised verification is undecidable in general. Nevertheless, expedient refinement heuristics are even more important in parameterised verification than in verifying fixed concurrent systems. Even if there exists a finite set of spotlight components that is sufficiently large for a definite result in verification, this set is generally more hard to detect for a parameterised system – because now the number of potential process instances of each class is *unbounded*, and thus, the refinement procedure might get lost in adding an infinite number of replications of one

process to the spotlight. In fact, using our original heuristic evaluation from Section 5.3 would suffer from exactly this problem. However, in the following we will see that heuristics specifically tailored to the verification of certain properties of class-wise symmetric systems can avoid such pitfalls.

Our running example refers to the verification of a liveness property of a parameterised message passing system. The heuristic approach that we present here is thus not only geared to *parameterisation*, but also to the *type of requirement* and to the *communication structure* of the system. In Figure 6.10 we have again an abstraction dependence graph corresponding to the initial abstraction of our message passing system. However, this time we distinguish *partner* and *competitor* dependencies (compare Section 5.3, Definition 5.6). For the sake of simplicity, the inherited dependencies of the spotlight process are omitted.

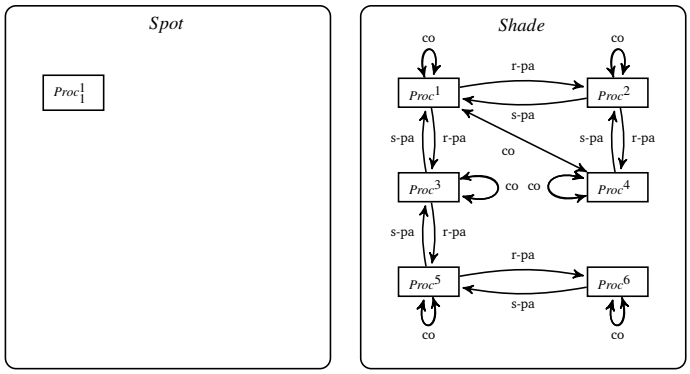

Fig. 6.10 Abstraction dependence graph corresponding to the spotlight abstraction $Spot = \{Proc_1^1\}$ and $Shade(Proc) = \{Proc^1, Proc^2, Proc^3, Proc^4, Proc^5, Proc^6\}$ of the class-wise symmetric message passing system $Sys_{MP} = \|_{m=1}^{6} (\|_{i \in PID_{N_m}^m} Proc_i^m)$. The edge labels denote the kind of dependency: *SendingPartner* is abbreviated by *s-pa*, *ReceivingPartner* is abbreviated by *r-pa* and *Competitor* is abbreviated by *co*.

Our current verification task refers to a global liveness property. Liveness of a certain class of processes in a parametrised message passing system inherently depends on the interplay between partners and between competitors. Processes from the same class are by nature competitors, which means that each process has potentially an unbounded number of competitors in a parameterised system. Hence, competitor dependencies are generally less distinctive structural features of parameterised message passing systems. Our first heuristic idea is thus to put particular emphasis on *partner* relationships when we analyse the dependencies within the shade. In our heuristic evaluation of refinement candidates we now use a modification *linkingShadeParam* of the original weight function *linkingShade* (compare Section 5.3.2, Table 5.4):

$$linkingShadeParam(v) :=$$

$$\sum_{v' \in Shade(Proc) \setminus \{v\}} \frac{size(v')}{s\text{-}paDistance(v',v) \cdot (instantiations(v')+1)}$$

Here $s\text{-}paDistance(v',v)$ returns the shortest directed path from v' to v in the subgraph of the abstraction dependence graph induced by edges within the shade that are labelled with *s-pa*. Moreover, $instantiations(v')$ returns the number of instantiations of v' in the spotlight. Hence, candidates that rely to a large extent on the assistance of sending partners from the shade are particularly expensive. The sub-function $instantiations(v')$ helps us to decrease the costs for dependencies to process classes of which we have already instantiations in the spotlight. However, we also use $instantiations(v)$ itself as a cost factor of process candidates v: The more instantiations of a certain class we have in the spotlight, the less is the expected gain of new information by adding another one from the same class.

Finally, we want to consider the benefit part of our heuristic evaluation. Our running example concerns the verification of a *liveness* formula. While *partners* are generally crucial for the *validation* of a liveness property, *competitors* are essential for its *refutation*. Naturally, the final outcome of a verification task is *not* known in advance. Thus, we assume that both, *partners* and *competitors* of spotlight processes are beneficial refinement candidates. For the evaluation of process candidates we use the following modification of the weight function *linkingSpot* (compare Section 5.3.2, Table 5.2):

$$linkingSpotParam(v) :=$$

$$\left|\left\{v' \mid v' \in Spot(Proc) \ \wedge \ Partner(v,v')\right\}\right|$$

$$+$$

$$\left|\left\{(v',v'') \mid v',v'' \in Spot(Proc) \ \wedge \ Partner(v',v'') \ \wedge \ \big(Comp(v,v') \ \vee \ Comp(v,v'')\big)\right\}\right|$$

Hence, we count the number of spotlight processes v' that are partners of the candidate v, and moreover, we count the number of partner relations (v',v") inside the spotlight that might be 'disturbed' by the candidate v as a competitor. – Our overall heuristic evaluation function for *process candidates* now corresponds to the following weighted composition of sub-functions:

$$h(v) \ :=$$

$$\underbrace{(\omega_1 \cdot occurrence(v) \ + \ \omega_2 \cdot linkingSpotParam(v))}_{benefit(v)}$$

$$-$$

$$\underbrace{(\omega_3 \cdot linkingShadeParam(v) \ + \ \omega_4 \cdot size(v) \ + \ \omega_5 \cdot instantiations(v))}_{cost(v)}$$

For *predicate candidates* we keep using the evaluation function introduced in Section 5.3.2. We now apply heuristic-guided abstraction refinement to our running example – the message passing system Sys_{MP}. After four iterations we get spotlight abstraction depicted in Figure 6.11.

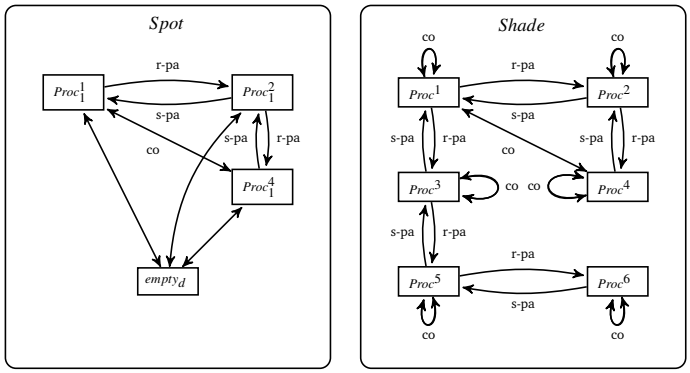

Fig. 6.11 Abstraction dependence graph corresponding to the spotlight abstraction $Spot(Proc) = \{Proc_1^1, Proc_1^2, Proc_1^4\}$ and $Shade(Proc) = \{Proc^1, Proc^2, Proc^3, Proc^4, Proc^5, Proc^6\}$, $Spot(Pred) = \{empty_d\}$ of the class-wise symmetric message passing system $Sys_{MP} = \|_{m=1}^{6} (\|_{i \in PID_{N_m}^m} Proc_i^m)$. For the sake of simplicity, the dependencies between the spotlight and the shade are omitted.

Based on our enhanced heuristic evaluation for process candidates one partner $Proc_1^2$ and one competitor $Proc_1^4$ of the initial process $Proc_1^1$ have been instantiated in the spotlight. The competitor $Proc_1^4$ is in turn a partner of $Proc_1^2$. All three processes affect and depend on the predicate $empty_d$, which has also been added to the spotlight. This abstraction is already precise enough for refuting the global CTL property $\bigwedge_{\langle i \rangle \in PID_{N_1}^1} \mathbf{AG}((pc_i = 3) \Rightarrow \mathbf{AF}(pc_i = 4))$ for the entire parameterised message passing system Sys.

Of course, the extended evaluation function is geared to a concrete verification problem, and thus, the successful verification in this case does not automatically prove the general efficiency of our extension. However, it demonstrates that our heuristic framework from Chapter 5 can be easily adapted for specific verification tasks, in particular for *parameterised verification*. Avoiding too many process instantiations of one class, and distinguishing different kinds of dependencies can even be considered as basic principles for an heuristic approach to abstraction refinement for parameterised systems. Concluding this chapter, we have shown that our heuristic-guided abstraction refinement framework is generally compatible with parameterised verification, which in several cases enables us to construct small abstractions that are precise enough for parameterised verification. In the next chapter, we will more extensively investigate the applicability of our verification framework, based on

an experimental evaluation. Beforehand, we take a look at related work on parameterised verification.

6.6 Related Work

Parts of our work introduced in this chapter have already been published in [119]. Moreover, our research on verifying parameterised systems via spotlight abstraction is connected to other approaches in a number of ways. In this section we summarise and extend our previous references to related works.

The *parameterised verification problem* has received considerable attention in research. Its undecidability was shown by Apt and Kozen in [7]. Nevertheless, several techniques have been proposed to bypass this issue. One way of approaching parameterised verification is to use *semi-automatic proof methods* based on theorem proving. The invisible invariants method proposed by Pnueli et al. [109] computes invariants on small instantiations of the parameterised system and proves that these are inductive on the entire system. McMillan's compositional reasoning [97] is another technique for parameterised verification that relies on inductive proofs. Furthermore, there exist a number of approaches based on *abstraction and model checking*. Baukus et al. [16] presents a technique for constructing abstract models of parameterised systems in the second order logic WS1S. Their approach requires manual intervention by a user in terms of defining abstraction relations. Regular model checking [3] and monotonic abstraction [2] are methods for automatically verifying ω-regular safety properties of parameterised systems. Both methods are based on automata-theoretic constructions and thus involve a high computational complexity.

Several other abstraction-based verification techniques for parameterised systems rely on *symmetry arguments*. Symmetry reduction [58, 105, 39] is a well-established method for reducing the state space complexity in temporal logic model checking. The general concept of symmetry reduction is to build equivalence classes of states that only differ in permutations of variable values. This method can be applied to any kind of system that exhibits some form of replication. Since fully symmetric and class-wise symmetric systems inherently consist of replicated processes, symmetry reduction can lead to substantial savings here. Thus, we selected this method as the basis for our approach to parameterised verification. From Clarke et al. [39] we have taken the idea of exploiting symmetry under permutations. However, Clarke et al. exploit symmetry in finite state systems, whereas our approach is tailored to systems with an unbounded number of processes. Similar to our work, Emerson and Kahlon [57] use symmetry arguments to reduce the verification of class-wise symmetric parameterised systems to the verification of small instances. Their method is complete for systems whose processes exhibit a particular type of transition guards. Moreover, it is restricted to the verification of safety

properties from a fragment of the temporal logic $CTL^*\backslash X$. The model checking procedure of [57] iterates over all instantiations up to a *cutoff* size – which is derived based on the structure of the system. Cutoffs for parameterised verification are also considered by Namjoshi in [103] who shows that that the cutoff technique is equivalent to determining a parameterised inductive invariant. Another cutoff-based approach to parameterised verification is presented by Kaiser et al. in [86] who propose a technique for dynamic cutoff detection during verification. The techniques counter abstraction by Pnueli et al. [110] and environment abstraction by Clarke et al. [35] are in a sense based on cutoffs too. Here the number of symmetric processes being in particular states is counted, where the counters are cut off at a certain number. The major difference between our approach and the cutoff-based methods [57, 103, 86] is that we start with an initial instantiation whose size is chosen based on the property to be checked. Moreover, we have no specific constraints concerning the transition guards and our approach is not limited to safety properties. The price that we pay is that our technique is incomplete, i.e. our abstraction refinement procedure might not terminate for all parameterised verification tasks. *Symmetry reduction combined with counterexample-guided abstraction refinement* has recently been considered by Donaldson et al. [53]. The authors introduce a symmetry-aware CEGAR technique for fully symmetric concurrent systems. Abstraction is based on a single representative process and a number of *mixed predicates*: boolean predicate expressions that refer to both shared and local variables at once. Refinement is performed like in classical CEGAR [34] based on spurious counterexamples. Symmetry permits to transfer reachability properties of the single process to the entire system. The proposed approach is more restrictive than our method. However, the integration of the concept of mixed predicates into our framework appears as a promising direction for future research. The previously mentioned technique monotonic abstraction [2] has also been integrated into a CEGAR framework [1]. Refinement in [1] is not based on adding new predicates but on introducing so-called *safety zones*: sets of configurations that satisfy certain requirements.

Our approach to parameterised verification works for fully and class-wise symmetric systems. Nevertheless, there exist a number of other notions of symmetry in concurrent systems as well as corresponding reduction techniques. In several parameterised systems the processes communicate in a ring. Classical examples are protocols for the dining philosophers problem. Such systems exhibit rotational symmetry, i.e. only circular process permutations are feasible. Parameterised verification of rotational symmetric systems is supported by the model checker SVISS [124], which however requires the specification of symmetry sets by a user. Many parameterised systems that consist of homogeneous processes are, strictly speaking, not symmetric because the processes execute id-sensitive operations. For instance, in Szymanski's mutual exclusion algorithm [117] and in Lamport's bakery algorithm [93] there exist operations that depend on the identifier of the executing process.

Counter abstraction-based approaches to the verification of parameterised systems with id-sensitive operations have been proposed by Pnueli et al. [110] and by Emerson and Wahl [59]. Both techniques are limited to id-sensitive operations of a certain kind. Spotlight abstraction, as we employ it in our framework, is not compatible with these different notions of symmetry in concurrent systems. Thus, an extension of our approach towards support for rotational symmetry and id-sensitive operations remains as future work. Finally, we want to mention that symmetry reduction is not only used in the context of concurrency. There also exist verification techniques that exploit data symmetry [43] or heap symmetry [84] in software systems.

Chapter 7
Implementation and Experimental Results

In the previous chapters, we conceptually introduced our approach to the verification of concurrent systems. We presented our heuristic framework for abstraction refinement and illustrated it by a number of simple examples. In this chapter, we describe the implementation of our framework. Moreover, we present two case studies that demonstrate the applicability of our approach for larger-scale systems. Within these case studies we show that our heuristic approach can significantly outperform naive refinement strategies. Moreover, we evaluate the suitability of certain weight configurations of our heuristic function for different verification tasks.

7.1 3Spot Verification Framework

We have implemented our approach to heuristic-guided abstraction refinement within the *3Spot verification framework*. 3Spot, initially introduced in [112], is a prototype tool for the verification of concurrent software systems. As input it takes a concurrent system written in a C-like syntax and a CTL formula over the systems variables and its control flow. For the input system nearly all control structures of the C language [88] are admissible. Originally, only the data types *boolean*, *integer* and *semaphore* where supported. Within this work we extended the input language to *counting semaphores*, *finite arrays* and *communication channels* (see Chapter 3 for a detailed description of the feasible input systems). 3Spot fully automatically checks whether the CTL property holds for the concurrent system and outputs the obtained result. In the following, we want to take a closer look at the steps between the input and the final output. In Figure 7.1 we can see the toolchain of the 3Spot verification framework.

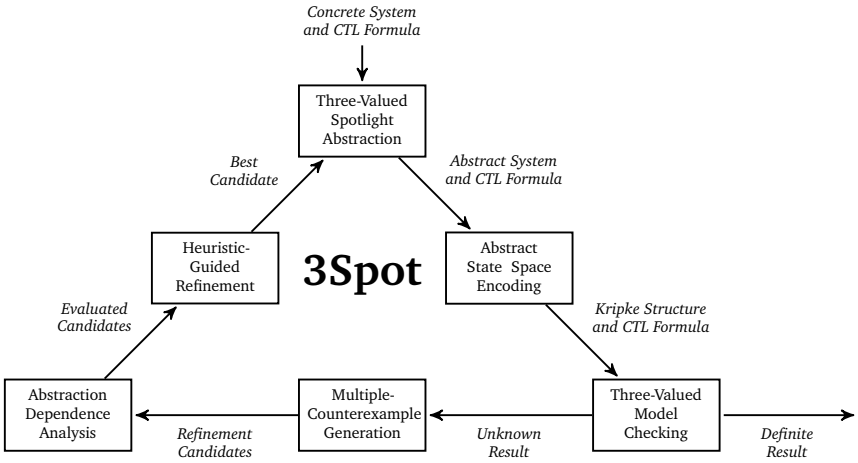

Fig. 7.1 Toolchain of the 3Spot verification framework.

In the first step, *three-valued spotlight abstraction* is applied to the input system (compare Sections 4.1 and 4.2): This requires an initial spotlight which is directly derived from the input CTL formula. The system is transformed into an equivalent control flow representation (compare Definition 3.5). Then an abstraction of the control flow representation is built by employing the theorem prover Z3 [102]. In the second step, a *state space encoding* of the abstract system is constructed (compare Definition 3.7). More precisely, a Kripke structure corresponding to the abstract systems state space is not explicitly built, but symbolically represented as a multi-valued decision diagram (MDD) [116]. In the third step, *three-valued model checking* is applied to the MDD-represented Kripke structure: The multi-valued decision diagram and the CTL formula are fed into the multi-valued symbolic model checker χChek [54]. χChek either returns a definite result that can be directly transferred to the original system, or it returns *unknown*. In the latter case, *multiple-counterexample generation* is employed (compare Section 5.2). The number of generated counterexamples can be specified by the user. A set of refinement candidates is derived from the counterexamples. In the subsequent step, an *abstraction dependence graph* is constructed for the current abstraction and the set of candidates (compare Section 5.3.1). The dependence structure of the abstraction is analysed and the candidates are evaluated with regard to their benefits and costs for refinement. Finally, *heuristic-guided refinement* determines the presumably best candidate and adds it to the spotlight (compare Section 5.3.2). This heuristic decision also incorporates user-specified weight parameters that allow to put particular emphasis on certain characteristics of the candidates. Now, the abstraction refinement loop starts again and iterates until a definite result in verification is achieved. The overall cycle runs fully automatic. The only task of the user is to select the inputs, chose a predefined

refinement heuristic, adjust the heuristic by setting the weight parameters, and finally, push the button. In each iteration the generated counterexamples and the constructed abstraction dependence graph are visualised so that the user can keep track of the heuristic refinement decisions. The graphical user interface of 3Spot is depicted in Figure 7.2.

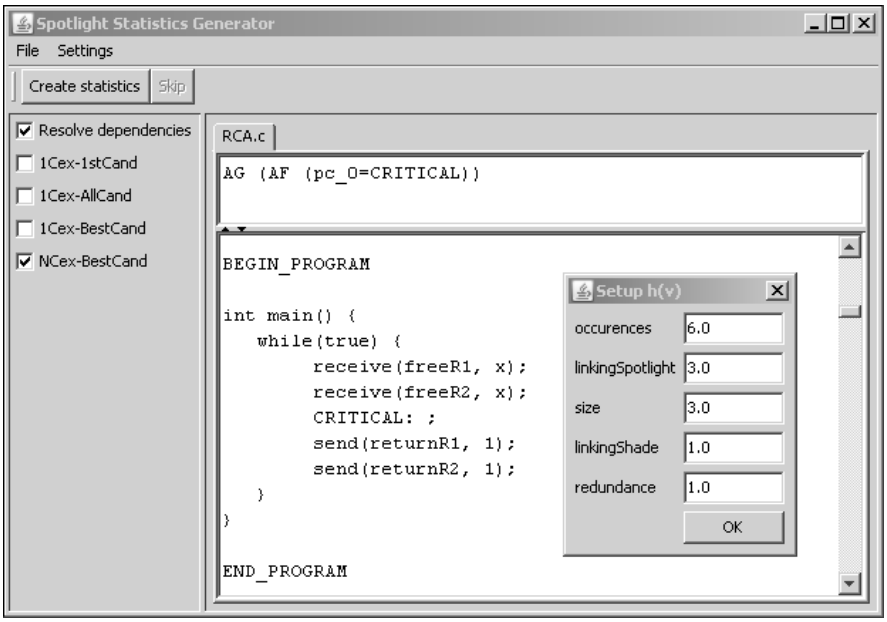

Fig. 7.2 Graphical user interface of the 3Spot verification framework.

Thus, 3Spot's graphical user interface displays the two inputs: the CTL formula and the program code of the concurrent system. The user can select the desired refinement heuristics via check boxes. Moreover, the weights of the evaluation function can be modified. A click on *'Create statistics'* then starts the verification run. The running abstraction refinement procedure is visualised by means of the iteratively generated abstraction dependence graphs. In Figure 7.3 we have an example of such a graph constructed by 3Spot.

We can identify the derived refinement candidates as a subset of the shade. Moreover, we can see their dependencies to the remaining components of the abstraction. In general, the visualisation of the constructed dependence graphs provides the user with a comprehensible view on the current stage of abstraction and the taken refinement decisions. Furthermore, it invites the user to experiment with the different heuristic parameters, and to visually keep track of the effects. In the subsequent section we will take a closer look at the selectable heuristics and at our experiments with 3Spot that we have conducted within this work.

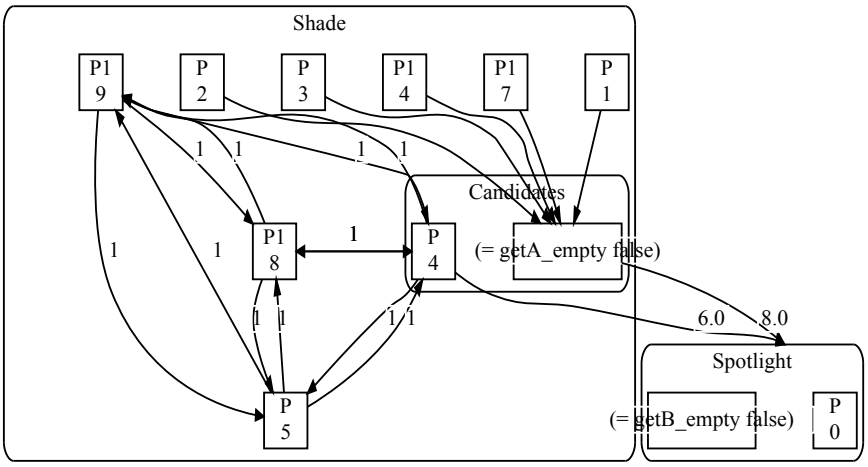

Fig. 7.3 Abstraction dependence graph constructed by 3Spot

7.2 Case Studies and Experimental Results

In Chapter 5 we extensively discussed the foundations of our heuristic framework for abstraction refinement. We considered several examples that illustrated the underlying ideas of our approach and demonstrated its general effectiveness for the verification of fixed-sized concurrent systems. Moreover, in Section 6.5 we showed that our heuristic approach can be also successfully applied to parameterised systems. However, the simple examples that we considered were geared towards descriptiveness, and thus, did not exhibit the complexity of real-life applications. Now, we want to show that our approach also performs well for larger systems.

While *uniform* concurrent systems can be usually efficiently verified based on symmetry arguments, the verification of *non-uniform* concurrent systems remains particularly challenging. The heterogeneous structure of such systems prevents the applicability of several well-established state space reduction techniques like counter abstraction [110] or symmetry reduction [39]. In contrast, the capability of *our* approach particularly arises from the effective exploitation of the – possibly non-uniform – structure of the considered system. In our case studies we therefore look at concurrent systems with arbitrarily chosen, and hence non-uniform dependencies. More specifically, we consider *multiple-resource allocation via message passing*, where we have two kinds of processes: *allocators* and *customers*. Each allocator manages the allocation of a *single* resource, whereas every customer requests *several* resources before entering its critical section. Moreover, allocators that provide a resource R_1 can in turn be customers of a resource R_2, i.e. they have to acquire R_2 first in order to provide R_1. This causes a high degree of transitive dependencies in our

systems. Allocators and customers communicate via message passing, i.e. each resource type is associated with a distinct tuple of channels. The customers' individual resource demands and the orders of requests are arbitrarily chosen. Hence, our systems exhibit a heterogeneous dependency structure. In Figure 7.4 we see an example of a fixed-sized multiple-resource allocation system.

$$freeR_1, returnR_1, \ldots, freeR_k, returnR_k : \textbf{channel}\ [1]\ \textbf{of integer}$$

$$\|_{i=1}^{k}\ Alloc_i :: \left[\begin{array}{ll} 1: & \textbf{loop forever do} \\ \left[\begin{array}{ll} 2: & \textbf{send}(freeR_i, \cdot) \\ 3: & \textbf{receive}(returnR_i, \cdot) \end{array}\right] & \end{array}\right]$$

$$\|\ Cust_1 :: \left[\begin{array}{ll} 1: & \textbf{loop forever do} \\ \left[\begin{array}{ll} 2: & \textbf{receive}(freeR_1, \cdot) \\ & \ldots \\ & \textbf{receive}(freeR_k, \cdot) \\ j: & \textbf{critical} \\ & \textbf{send}(returnR_1, \cdot) \\ & \ldots \\ m: & \textbf{send}(returnR_k, \cdot) \end{array}\right] & \end{array}\right] \|\ Cust_2 :: \left[\begin{array}{ll} 1: & \textbf{loop forever do} \\ \left[\begin{array}{ll} 2: & \textbf{receive}(freeR_k, \cdot) \\ & \ldots \\ & \textbf{receive}(freeR_1, \cdot) \\ j: & \textbf{critical} \\ & \textbf{send}(returnR_k, \cdot) \\ & \ldots \\ m: & \textbf{send}(returnR_1, \cdot) \end{array}\right] & \end{array}\right]$$

Fig. 7.4 Message passing system $Sys_{MRA} = \|_{i=1}^{k}\ Alloc_i \|\ Cust_1 \|\ Cust_2$ over $Var = \{freeR_1, returnR_1, \ldots, freeR_k, returnR_k\}$ with multiple-resource allocation.

In the system Sys_{MRA} we have k allocators and two customers. Each allocator $Alloc_i$ manages a resource R_i. By sending on the channel $freeR_i$ the allocator makes the resource available in the system. As we can see, the customer $Cust_1$ attempts to acquire the resources in the order $R_1, \ldots, R_k$, whereas $Cust_2$ requests the resources in the reverse order. Once a customer has acquired all demanded resources it can enter a critical section. Afterwards the customer releases its acquired resources by sending on the channels $returnR_i$, $1 \leq i \leq k$. The allocators then receive their managed resources back and make them again available in the system. Sys_{MRA} is a basic example for a message passing system with multiple resource allocation. As we have mentioned before, such systems may also consist of *several* allocators per resource type, moreover, customers do not necessarily request *all* resources that are offered in the system, and allocators may be *in turn* customers of other resources. Thus, Sys_{MRA} with $k = 1$, i.e. a single allocator, is the smallest system that we consider in our first case study. We then proceed to larger-scale systems by introducing more resources, more allocators and more customers. As a measure for the *degree of dependence* in a system we use the number of edges in the corresponding dependence graph. In Figure 7.5 we have the dependence graph corresponding to Sys_{MRA} with $k = 1$. As we can see, the degree of dependence is 6.

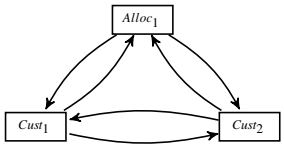

Fig. 7.5 Dependence graph corresponding to $Sys_{MRA} = Alloc_1 \parallel Cust_1 \parallel Cust_2$.

7.2.1 Case Study I: Naive vs. Enhanced Refinement

The requirement that we consider in our first case study is the CTL liveness formula $\mathbf{AG}(\mathbf{AF}(pc_{Cust_1} = j))$, i.e. we check whether the customer $Cust_1$ will continuously reach the critical section. Although we perform verification under the assumption of fairness, such a requirement can be potentially violated in multiple resource allocation systems. Customers that compete for the same types of resources may prevent each other from entering the critical section. Hence, proving or disproving liveness properties may necessitate to draw a larger set of mutually dependent processes (and corresponding predicates) into the spotlight. Our abstraction refinement framework aims at heuristically discovering the minimal set of processes and predicates that is sufficient for a definite result in verification. In this case study we thus compare naive refinement approaches with our enhanced approach based on the heuristic evaluation of refinement candidates. More precisely, we take a look at the following heuristics for counterexample-guided abstraction refinement:

1CEX-1STCAND (naive)	generate *one* unconfirmed counterexample π select *first* candidate v along π for refinement
1CEX-ALLCAND (naive)	generate *one* unconfirmed counterexample π select *all* candidates $v_1, \ldots, v_m$ along π for refinement
1CEX-BESTCAND (enhanced)	generate *one* unconfirmed counterexample π select *best* candidate v for refinement wrt. the evaluation function h
NCEX-BESTCAND (enhanced)	generate N unconfirmed counterexamples $\pi_1, \ldots, \pi_N$ (if existing) select *best* candidate v for refinement wrt. the evaluation function h

Table 7.1 Heuristics for counterexample-guided abstraction refinement.

The first two heuristics correspond to the naive approaches: We generate one unconfirmed counterexample and then select the first encountered candidate (1CEX-1STCAND), or alternatively, all candidates (1CEX-ALLCAND) for refinement. In our enhanced heuristics we use the evaluation function h for selecting the presumably best candidate, either for a single counterexample (1CEX-BESTCAND) or with multiple counterexample-generation (NCEX-BESTCAND).

Remember that heuristic evaluation function $h : Candidates \to \mathbb{R}$ corresponds to a weighted composition of sub-functions with

$$h(v) :=$$

$$\underbrace{(\omega_1 \cdot occurrence(v) + \omega_2 \cdot linkingSpot(v))}_{benefit(v)}$$

$$-$$

$$\underbrace{(\omega_3 \cdot linkingShade(v) + \omega_4 \cdot size(v) + \omega_5 \cdot redundancy(v))}_{cost(v)}$$

for $v \in Candidates$. A detailed description of the sub-functions of h can be found in Section 5.3.2. However, so far, we have not discussed the weight parameters $\omega_1, \ldots, \omega_5$.

Our choice of the weights for the first case study relies on a number of facts and observations: The sub-functions of h generally exhibit different ranges of values. For a candidate v the value of $linkingShade(v)$ is typically considerably higher than the values of the remaining sub-functions. We further observed that the likeliness for a (relatively) high value of $occurrence(v)$ increases with the number of generated counterexamples. The same holds for $linkingSpot(v)$ and the number of components in the spotlight, for $linkingShade(v)$ and the degree of dependence within the shade, as well as for $size(v)$ and the number of channels a process candidate v communicates on. The value of $redundancy(v)$ is generally small in multiple resource allocation systems without additional local variables. Moreover, the values obtained for $linkingSpot(v)$ and $linkingShade(v)$ are commonly subject to significantly higher fluctuations than the values of $occurrence(v)$, $size(v)$ and $redundancy(v)$.

For our first case study we thus selected a weight configuration based on the following ideas: On the one hand, the different ranges of values should be equalised, i.e. a balance between the benefit-side and the cost-side should be achieved. On the other hand, there should be some extra emphasis on the two potentially most crucial sub-functions $linkingSpot(v)$ and $linkingShade(v)$. This resulted in the following weighting for h:

$$h(v) :=$$

$$\underbrace{(6 \cdot occurrence(v) + 6 \cdot linkingSpot(v))}_{benefit(v)}$$

$$-$$

$$\underbrace{(3 \cdot linkingShade(v) + 1 \cdot size(v) + 1 \cdot redundancy(v))}_{cost(v)}$$

Table 7.2 now shows the experimental results of our case study I:

	System					Abstraction		
heuristic	resources	customers	allocators	dependencies	requirement	$\|Spot(Proc)\|$	$\|Spot(Pred)\|$	time
1Cex-1stCand	1	2	1	6	$liveness, false$	3	1	0.74s
(naive)	2	5	3	36	$liveness, true$	6	1	6.94s
	3	6	4	48	$liveness, false$	6	1	8.83s
	4	8	8	76	$liveness, true$	–	–	OOM
	8	15	10	126	$liveness, false$	–	–	OOM
1Cex-allCand	1	2	1	6	$liveness, false$	3	1	0.67s
(naive)	2	5	3	36	$liveness, true$	5	2	7.06s
	3	6	4	48	$liveness, false$	5	4	195s
	4	8	8	76	$liveness, true$	5	4	307s
	8	15	10	126	$liveness, false$	–	–	OOM
1Cex-bestCand	1	2	1	6	$liveness, false$	3	1	1.39s
(enhanced)	2	5	3	36	$liveness, true$	5	2	16.4s
	3	6	4	48	$liveness, false$	6	2	27.3s
	4	8	8	76	$liveness, true$	5	4	83.9s
	8	15	10	126	$liveness, false$	–	–	OOM
2Cex-bestCand	1	2	1	6	$liveness, false$	3	1	1.67s
(enhanced)	2	5	3	36	$liveness, true$	3	2	6.47s
	3	6	4	48	$liveness, false$	3	2	6.52s
	4	8	8	76	$liveness, true$	3	2	8.55s
	8	15	10	126	$liveness, false$	4	3	57.5s
3Cex-bestCand	1	2	1	6	$liveness, false$	3	1	1.91s
(enhanced)	2	5	3	36	$liveness, true$	3	2	6.82s
	3	6	4	48	$liveness, false$	3	2	6.90s
	4	8	8	76	$liveness, true$	3	2	8.49s
	8	15	10	126	$liveness, false$	4	3	69.9s

Table 7.2 Comparison of the performance in verification under naive and enhanced refinement heuristics.

Hence, we compared five heuristics: the two naive ones, and our weighted *best candidate* heuristic with one, two, and three generated counterexamples; and, we considered five different multiple-resource allocation systems with an increasing number of resources, processes and dependencies. For two of these systems the liveness requirement $\mathbf{AG}(\mathbf{AF}(pc_{Cust_1} = j))$ holds (*liveness,true*), whereas for three systems the requirement is not satisfied (*liveness,false*). For each heuristic and each system we measured the size of the final abstraction and the required verification time.

For the two smallest systems the naive heuristics can slightly outperform our enhanced heuristics with regard to the verification time. However, this does not hold for the size of the final abstraction. Here our enhanced heuristics perform minimally better or at least equally good as the naive ones. The difference in verification time is caused by the additional computations that come along with our *best candidate* heuristics: Multiple counterexamples have to be generated and an abstraction dependence graph has to be built and analysed in each iteration. The fact that the employment of our enhanced heuristics does not lead to a significant reduction of the size of the final abstraction can be explained as follows: The two smallest systems exhibit – in relation to their size – a comparatively high degree of dependence, and thus, verification generally requires to draw a large part of the overall system into

the spotlight. Hence, there is only little scope for optimising the size of the abstraction.

Nevertheless, for the larger systems (with three resources and more) the additional computations of the enhanced heuristics pay off: Verification under 1CEX-BESTCAND already entails a slightly better performance than the naive approaches. The use of the *best candidate* heuristics with *multiple-counterexample generation* leads to significantly smaller abstractions and clear speed-ups in time. With these enhanced heuristics we are even successful and fast when verification under the other heuristics runs out of memory (OOM). We can observe that our advanced approach to abstraction refinement facilitates the detection of small abstractions that are already precise enough for definite results in verification. The heuristic evaluation of refinement candidates in terms of dependencies to the spotlight and to the shade allows us to identify valuable refinement steps and to avoid unfavourable directions. Moreover, we see that our approach can lead to improvements for *both* the *validation* and the *refutation* of local liveness requirements. With multiple-counterexample generation we have an additional source of refinement candidates, and thus, a greater scope for heuristic decisions. In our framework multiple counterexamples also serve as the basis for *region summarisation* (compare Section 4.2.3). Thus, it is not surprising that the *best candidate* heuristics with two and three generated counterexamples entail the best performance in our case study. However, the generation of a third counterexample yields no further improvement in comparison to the heuristic 2CEX-BESTCAND. We conjecture that this may hold for the scale of systems that we considered in our first case study – but the verification of larger systems with more dependencies may profit from a greater number of generated counterexamples.

7.2.2 Case Study II: Weight Configurations

In our second case study we investigated the impact of distinct weight parameters on certain verification tasks. For this purpose, we looked at a number of local requirements that could not be checked for larger-scale resource allocation systems under the naive heuristics. We employed the *best candidate* heuristic in these verification tasks and compared the performance of our framework under different weight configurations. The first configuration that we considered was the one that we introduced in our first case study, which we will from now on call the *balanced* configuration. Second, we regarded a setting where most of the emphasis was put on the benefit-side, called the *benefit-focused* configuration. And analogously, we took a look at a *cost-focused* configuration with particular emphasis on the costs for refinement. The requirements that we checked were defined as CTL formulae of the form $\mathbf{AG}(\mathbf{AF}(pc_{Cust_1} = j))$ and $\mathbf{AG}\neg((pc_{Cust_1} = j) \wedge (pc_{Cust_2} = j))$, i.e. we considered

both liveness and safety properties. We verified a number of larger multiple-resource allocation systems that satisfied the requirements as well as systems that violated the requirements. An excerpt of the experimental results of our second case study is shown in Table 7.3 at the end of this section.

In a series of experiments we observed some interesting correlations between the chosen configurations and the performance of verification. Moreover, we identified two typical pitfalls in heuristic-guided refinement. The first pitfall particularly emerges under the benefit-focused configuration. It can be described by the following phenomenon: Despite the use of heuristics, a number of already selected refinement candidates may turn out to be unrewarding for a definite result in verification. However, those inexpedient predicates and processes that are now part of the spotlight bias the benefit evaluation in future refinement iterations. Instead of adding truly rewarding components, more and more candidates are selected that fit to the unrewarding ones in the spotlight. One can imagine that this *benefit pitfall* can easily cause an unnecessary explosion of the state space. – Of course the reverse effect is also possible: A number of good refinement decisions in the past facilitate expedient decisions in the future. However, heuristics inherently do not involve optimality, and therefore, a proper approach to heuristic-guided refinement should also feature the ability to compensate inexpedient decisions in future iterations.

The second pitfall that we encountered in our experiments is cost-related, it can be exemplified as follows: In many cases the selection of an expensive candidate is an inevitable prerequisite for a definite result in verification. However, due to its high cost value several cheaper candidates may be preferred in the refinement decisions. Thus, the abstraction is at first unnecessarily enlarged by a number of lightweight components until the expensive but required candidate is finally added to the spotlight. As we can see, the cost-side of our heuristic evaluation, although generally helpful for avoiding too expensive directions in refinement, can be itself a *cost pitfall*.

Our experiments revealed that heuristic-guided abstraction refinement under the benefit-focused configuration is generally more prone to the benefit pitfall than the balanced configuration. The same holds for the cost-focused configuration and the cost pitfall. However, we also observed that the weight configuration by itself is not the only decisive factor for the success or fail in verification – the type of requirement and the fact whether it actually holds or not are additional influencing factors. In particular, we experienced that, contrary to our initial expectations, the balanced configuration is not necessarily the most confident guarantor for compensating inexpedient refinement decisions and avoiding the aforementioned pitfalls. Subsequently, we want to discuss our observations in more detail.

Experiments on *liveness refutation* for multiple-resource allocation systems showed that heuristic-guided abstraction refinement under the *cost-focused* configuration performs best for this type of verification task. Heuristically detecting liveness counterexamples requires the selection of a number of con-

tinuously interacting processes that are capable of causing resource starvation for a certain customer. The *cost-focused* configuration facilitates to choose processes that are preferably independent from the remaining shade. This frequently leads to a comparatively small number of interacting processes in the spotlight that suffice for a liveness counterexample. On the contrary, liveness refutation under the *benefit-focused* configuration (and interestingly, also under the *balanced* configuration) commonly suffered from the benefit pitfall.

Liveness validation generally demands more effort than refutation. A liveness property has to hold under all possible computations of the system. Validating such a property typically involves the addition of a larger cluster of mutually dependent processes and their associated predicates to the spotlight. We experienced that the selected weight configuration used for liveness validation affected the order in which candidates were added to the abstraction. In some cases this led to more efficiency under the *cost-focused* configuration. In other cases the performance under the *benefit-focused* configuration was better, and again, the performance results obtained under the *balanced* configuration were very similar to the ones under *benefit-focused* configuration. Thus, our experiments did not hint at a generally most suitable weight configuration for liveness validation.

In the verification of safety requirements we made the following observations: Counterexample detection for safety properties performed best under the *benefit-focused* configuration, whereas the *cost-focused* configuration often suffered from the cost pitfall. This is in accordance with the fact that a safety counterexample corresponds to a finite computation that ends in an error state. Processes involved in such a computation usually contribute to the counterexample with a small number of executed operations, which also implies that only a small part of their dependencies are relevant for this counterexample. Hence, for the *refutation* of safety properties the costs (or, more specifically, the shade dependencies) of refinement candidates are rather negligible. In line with this, the performance results obtained under the *balanced* configuration were located between the ones under the *benefit-focused* and the *cost-focused* configuration.

Our experiments on *safety validation* revealed no clearly best weight configuration. We experienced that sometimes cost aspects and sometimes benefit aspects were more crucial for efficiency in such verification tasks. Safety validation under the *balanced* configuration involved a medium performance. In most cases the performance results under one of the two unbalanced configurations were significantly better.

As a first conclusion, we can state that the cost-focused weight configuration is particularly suited for the detection of liveness counterexamples, whereas the benefit-focused configuration is particularly suited for detecting safety errors. However, the final outcome in verification is typically not known in advance. In case the considered property is not refutable, verification under one of the unbalanced weight configurations can easily end up in either the

benefit or the cost pitfall. On the contrary, verification under the balanced configuration is slightly more capable of avoiding these pitfalls, but we also experienced that its performance is only in a few cases the best among the three compared configurations. Our experiments further showed that the balanced configuration does not necessarily lead to a real balance between benefit and cost aspects in the heuristic decisions. We observed that verification under the balanced configuration leads in many cases to similar refinement decisions as under the benefit-focused configuration – and in just as many other cases to similar decisions as under the cost-focused configuration.

These rather unsatisfactory results in terms of a best suited weight configuration for heuristic-guided abstraction refinement encouraged us to extend our second case study by another configuration. As a new approach to more balance in heuristic selections we decided for a weight configuration that dynamically changes during verification. We experimented with the simplest variant of such a dynamic weighting: an *alternation* between the benefit-focused and the cost-focused configuration from iteration to iteration. Verification under the alternating configuration yielded an exceedingly good performance. For all types of verification tasks (i.e. liveness/safety refutation/validation) this dynamic approach to heuristic-guided abstraction refinement involved an above-average performance in comparison with the other configurations, which can be also seen in Table 7.3 where an excerpt of our experimental results is shown.

The alternating configuration facilitates to avoid the pitfalls in heuristic-guided abstraction refinement more effectively than the (static) balanced configuration. It ensures that refinement decisions are not narrowed down to single directions that finally turn out to be inexpedient, and moreover, it prevents that expedient but costly refinement candidates are permanently ignored. Our experiments revealed that the alternating configuration does not necessarily involve the performance optimum among all compared configurations for each individual verification task. In a number of cases the size of the final abstraction and the required verification time under the alternating configuration is slightly greater than under the respective best performing configuration. Nevertheless, we observed that verification under the alternating configuration performs at least close to the optimum, and moreover, rarely suffers from any pitfall.

Concluding our second case study, we can state that the efficiency of heuristic-guided abstraction refinement can be significantly influenced by the selection of the weight configuration. We observed that the benefit-focused configuration is particularly suited for the detection of safety counterexamples, whereas the cost-focused configuration facilitates liveness refutation. In certain cases, e.g. in early stages of the development of a system, the presence of errors can be presumed, which makes the selection of one of the aforementioned configurations for safety/liveness refutation advisable. However, verification is commonly performed under the assumption that it is uncertain whether the desired requirement holds for the considered system or

	System					Abstraction		
configuration	resources	customers	allocators	dependencies	requirement	$\|Spot(Proc)\|$	$\|Spot(Pred)\|$	time
balanced	3	4	4	34	$liveness, false$	≥ 5	≥ 4	OOM
	4	7	11	90	$liveness, false$	≥ 5	≥ 4	OOM
	4	10	9	98	$liveness, true$	4	4	116s
	6	13	8	110	$liveness, true$	4	3	81.8s
	5	8	6	44	$safety, false$	5	2	10.6s
	7	9	11	126	$safety, false$	6	3	7.45s
	6	12	6	88	$safety, true$	4	3	32.4s
	6	13	10	140	$safety, true$	7	4	321s
benefit-focused	3	4	4	34	$liveness, false$	5	3	405s
	4	7	11	90	$liveness, false$	≥ 5	≥ 3	OOM
	4	10	9	98	$liveness, true$	4	4	145s
	6	13	8	110	$liveness, true$	4	3	82.5s
	5	8	6	44	$safety, false$	5	1	4.88s
	7	9	11	126	$safety, false$	6	3	7.42s
	6	12	6	88	$safety, true$	5	3	50.8s
	6	13	10	140	$safety, true$	7	3	133s
cost-focused	3	4	4	34	$liveness, false$	4	2	8.28s
	4	7	11	90	$liveness, false$	5	3	324s
	4	10	9	98	$liveness, true$	4	5	1178s
	6	13	8	110	$liveness, true$	4	2	49.0s
	5	8	6	44	$safety, false$	5	2	10.7s
	7	9	11	126	$safety, false$	≥ 8	≥ 3	OOM
	6	12	6	88	$safety, true$	4	1	2.86s
	6	13	10	140	$safety, true$	≥ 8	≥ 5	OOM
alternating	3	4	4	34	$liveness, false$	4	3	23.1s
	4	7	11	90	$liveness, false$	5	3	308s
	4	10	9	98	$liveness, true$	4	4	146s
	6	13	8	110	$liveness, true$	4	2	49.1s
	5	8	6	44	$safety, false$	5	2	10.2s
	7	9	11	126	$safety, false$	7	3	34.7s
	6	12	6	88	$safety, true$	4	2	4.14s
	6	13	10	140	$safety, true$	7	4	267s

Table 7.3 Comparison of the performance in verification under the weight configurations *balanced*, *benefit-focused*, *cost-focused* and *alternating*.

not. Our experiments revealed that an alternation between benefit-focused and cost-focused refinement decisions is a generally well-performing approach to heuristic-guided abstraction refinement – for refutation as well as for validation. Thus, the alternating configuration provides a promising starting point for arbitrary verification tasks. Should verification nevertheless run out of memory, there still remains the opportunity to repeat verification under a different configuration.

Our two case studies have demonstrated the applicability of our heuristic-guided abstraction refinement technique for the verification of larger-scale concurrent systems (with up to 25 processes and 140 dependencies). In addition, we have seen that the choice of a reasonable weight configuration for the heuristic evaluation function can significantly increase the efficiency of verification. Nevertheless, within this work we could of course not evaluate every facet of our approach in a comprehensive case study, and besides, our experimental capabilities were subject to a number of technical limitations. In the subsequent section we will therefore discuss some noteworthy aspects of our approach that have not been covered in our case studies.

7.3 Discussion

The 3Spot verification framework that we used for our experiments is still a prototype tool. Although it accepts arbitrarily large parallel compositions of processes as an input, the maximum size of the constructed *abstractions* is strongly restricted. As we have seen in our case studies, abstractions with more than eight processes and more than five predicates are not manageable for 3Spot. The built-in model checker χChek typically runs out of memory when exploring abstract state spaces of this magnitude. Thus, the main challenge in our experiments was to heuristically detect abstractions that were on the one hand small enough to be manageable for 3Spot, and on the other hand precise enough for a definite result in verification. Of course, one can imagine that for an unbounded number of verification tasks it is impossible to detect a *small* abstraction that suffices for a definite answer in verification. Therefore, in our experiments a definite result was only achievable for verification tasks where such small and precise abstractions actually exist.

Nevertheless, we are convinced that, given a more efficient implementation of a multi-valued model checker, our approach to heuristic-guided abstraction refinement would also perform very well for more complex verification tasks. This would also enable us to employ dynamic weight configurations that are currently not applicable: So far, we deal with verification runs with usually not more than 15 refinement iterations. The capability to handle more complex abstractions would also involve a generally larger number of refinement steps. Hence, instead of a simple alternation between benefit-focused and cost-focused refinement decisions, distinct *weighting phases* could be introduced and the dynamic evolvement of the weight parameters could be e.g. controlled by a sine-like function. Such an advanced concept for a dynamically weighted heuristic evaluation could additionally help to prevent the refinement procedure from running into unfavourable directions.

In our two case studies, we restricted ourselves to the verification of local properties of multiple-resource allocation systems. However, in single experiments we have also considered other types of systems and parameterised verification tasks. While communication in multiple-resource allocation systems solely rely on message passing, we have also experimented with heuristic-guided abstraction refinement for shared variable concurrent systems. Here we also achieved significant savings in verification time and size of the final abstraction when we employed the advanced heuristics instead of a naive approach refinement. Moreover, we observed that for this type of systems the efficiency of verification can be further improved by putting additional emphasis on the *redundancy* part in the heuristic evaluation of refinement candidates. In comparison to communication channels, shared variables can occur in arbitrary operations, e.g. in guards of *if*-branches and *while*-loops as well as in arbitrary assignments. Hence, the number of possible predicates over shared variables is generally higher than the number of possible predicates over communication channels. This fact also involves an enhanced

risk of adding too many dispensable, or rather, *redundant* predicates over the same shared variable to the abstraction. Thus, the improved verification results under a redundancy-emphasised weight configuration comply with our expectations. We also conducted some experiments on heuristic-guided abstraction refinement for parameterised verification. As we already mentioned in Section 6.5, the heuristic evaluation function that we employed for abstraction refinement of parameterised systems is slightly different to the one we used for fixed-sized concurrent systems. In our experiments we frequently achieved a better performance in parameterised verification when we put additional emphasis on the benefit-side in our refinement decisions. This is in line with the fact that in parameterised verification shade dependencies (i.e. costs) are consistently very high, and thus, less significant for expedient refinement decisions.

In conclusion, our experiments have revealed a number of interesting and promising results with regard to the improvement of verification by the use of heuristics. The presented case studies have demonstrated the practical applicability of our technique. We were even able to successfully verify a number of larger-scale concurrent systems, provided that there existed abstractions that were precise enough for a definite result *and* small enough for the capabilities of our prototype implementation. Nevertheless, we see a great potential for future enhancements and extensions of our framework. In the eighth and final chapter of this thesis we will conclude our work and address possible directions for future research.

Chapter 8
Conclusion

In this chapter we conclude the thesis with a summary and a critical discussion of the achieved results. Moreover, we propose promising directions for future research.

8.1 Summary

Within this thesis, we introduced an abstraction refinement-based verification framework for concurrent software systems. The major challenge in software verification is the so-called *state explosion problem*: Verifying a software system generally involves an exhaustive exploration of the corresponding state space. The state space complexity grows exponentially with the size of the system or, more specifically, with the number of processes in a concurrent system. Hence, the straightforward verification of real-life software commonly fails due to unaffordable computational demands. *Abstraction* is a well-established approach to the state explosion problem in verification. The basic idea is to construct an abstract, and thus small system model, that preserves the validity of certain properties of interest. In case the model turns out to be too coarse for a definite result in verification, the abstraction is iteratively refined until an adequate level of precision has been reached. – Abstraction is not a unitary approach. In fact, there exists a wide range of different abstraction techniques, each geared towards a specific facet of software systems like data, control or interleaving. The effectiveness of abstraction-based verification thus essentially depends on the choice of an adequate way of abstraction. – *Refinement* is an equally crucial task. Abstractions are refined by gradually adding more details to the system model. Hence, a sequence of advantageous refinement decisions may entail a definite result in verification – whereas unfavourable decisions may unnecessarily enlarge the model and thus lead to state explosion. Abstract models commonly reveal several directions for refinement which makes the selection of the 'right' details particularly challenging.

In our verification framework we focused on *concurrent systems*, which are of increasing importance in the fields of distributed computing and network technologies. However, verifying concurrent systems is exceedingly difficult. Concurrency is one of the main contributors to state explosion in software systems. We approached this problem by *spotlight abstraction*, a state space reduction technique that is specifically tailored to concurrent systems. The basic idea of spotlight abstraction is to set a so-called *spotlight* on processes of particular interest while the remaining system is kept in the *shade*. Now, the spotlight processes are thoroughly considered when constructing an abstract system model, whereas processes in the shade are summarised into one abstract component. We enhanced this concept by introducing *shade clustering*, a technique that summarises similar shade processes into groups such that fewer details about them get lost. Spotlight abstraction preserves several local properties of concurrent systems while the number of possible interleavings is considerably reduced. – However, concurrency is not the only source of state explosion. Large variable domains likewise contribute to an exponential growth of the state space. We therefore combined spotlight abstraction with *predicate abstraction*. Here the system under consideration is approximated by an abstract model in which the original variables are replaced by atomic predicates over these variables. The predicates can be restricted to certain variables of interest in the verification task. Moreover, the domain of predicates can be substantially narrowed down in relation to the domain of the system variables, which involves a significant reduction of the state space. While most existing abstraction frameworks are based on a boolean domain for predicates, we use a *three-valued* domain with the values *true*, *false* and *unknown*. The additional truth value *unknown* enables us to model the inherent loss of information due to abstraction in a very natural way. Furthermore, in comparison to boolean abstractions our approach is capable of preserving both existential and universal system properties. All definite verification results obtained on a three-valued model can be directly transferred to the original system. Only *unknown* results necessitate abstraction refinement.

The combination of spotlight abstraction and three-valued predicate abstraction generally enables us to verify concurrent systems on small abstract models. However, finding the right degree of abstraction is highly nontrivial – abstract system models can be too imprecise or, on the other hand, too complex for a definite result in verification. In our framework we approached this problem by *counterexample-guided abstraction refinement* (CEGAR). Starting with a very coarse initial model, CEGAR iteratively refines the abstraction based on a counterexample analysis. Counterexamples are paths in the abstract model that disprove the property under consideration. These paths may correspond to a real computation of the system, or they may be *unconfirmed*. While the former case involves the definite refutation of the property, the latter demands abstraction refinement. Unconfirmed counterexamples hint at a number of possible refinement steps that may help to either confirm or eliminate the indefinite path in the abstract model. However, relying on the

guidance of counterexamples can be full of pitfalls. The refinement procedure can easily get lost in concretising inexpedient paths without converging to a definite result in verification. Our combined abstraction technique adds another facet to refinement: Counterexamples may point to predicates as well as to processes as refinement candidates. This increases the opportunities for both expedient and unfavourable refinement steps, and thus makes an elaborate approach to refinement even more important. We therefore introduced *heuristic-guided abstraction refinement* in our verification framework. Based on an *abstraction dependence analysis* the benefits and costs of potential refinement steps are heuristically evaluated and in each iteration the most beneficial candidate is added to the abstraction. Benefits arise from the gain of definite information that can be obtained by adding a candidate to the abstraction, whereas costs are mainly induced by the dependencies of a candidate to the shade. The refinement candidates are derived by *multiple counterexample-generation*. This gives us more candidates per iteration, and hence, more capabilities for heuristic decisions. Our heuristic approach enables us to guide the refinement procedure in expedient directions, and thus to construct small abstract models of concurrent systems that are precise enough for a definite result in verification.

Spotlight abstraction, one of the main concepts of our framework, is generally restricted to the verification of local properties of concurrent systems. Hence, it allows for the validation of properties that refer to a finite subset of the systems processes – though, due to shared variables or message passing, such properties may also depend on the behaviour of the remaining system. Within this thesis, we demonstrated that combining spotlight abstraction with *symmetry reduction* permits us to use our framework in the context of *parameterised verification* as well. Parameterised systems consist of an arbitrary number of processes that can be partitioned into a fixed number of classes of homogeneous or rather *symmetric* processes. Verifying a parameterised system is concerned with checking whether a global property, i.e. a property that is universally quantified over the processes of certain symmetry classes, holds for *all* possible instantiations of the system. – Parameterised verification is undecidable in general. However, spotlight abstraction together with symmetry arguments enables us to obtain definite verification results in several cases. We proved that the inherent symmetry of parameterised systems can be exploited in order to reduce global verification tasks to local ones. Moreover, we showed that the spotlight principle allows for the summarisation of arbitrary-sized instantiations of a parameterised system in one abstract model. Finally, heuristic-guided abstraction refinement can be applied to such a model, which gives us a verification procedure for parameterised systems. Due to the undecidability of parameterised verification this procedure does not terminate in general. Nevertheless, we demonstrated that a number of parameterised verification tasks can be efficiently accomplished through this approach.

We implemented our fully automatic verification framework within the tool *3Spot* and evaluated it in two case studies. Here we considered the verification of safety and liveness requirements of concurrent systems. We showed that the core of our framework – a modular heuristic evaluation function for refinement candidates – can be flexibly adjusted to the specific characteristics of the underlying verification task. The experimental results revealed that our heuristic approach to the verification of concurrent systems can, in many cases, significantly outperform classic refinement approaches in both size of the final abstraction and verification time.

8.2 Discussion

Although our approach to the verification of concurrent systems revealed very promising results, we assume that there exists a high potential for further enhancements. In this section we want to discuss and evaluate alternatives to the techniques and concepts that we have integrated into our framework. One of the major questions that we addressed in this thesis was how abstraction-based verification for concurrent systems could be improved by the use of heuristics. Hence, the first step towards answering this question was to choose an adequate way of abstraction for concurrent systems. Preliminary work [112] provided us with a general frame based on spotlight abstraction and three-valued predicate abstraction. This combination turned out to be highly effective in reducing the state space complexity induced by concurrency and data. Nevertheless, the choice of a *three-valued* domain for predicates requires some additional comments: Since the truth value *unknown* is an integral part of the spotlight principle, we deliberately decided to use an abstraction that exceeds the boolean domain. However, this limited our options for abstraction refinement techniques. Interpolation-based refinement [96], a well-established concept for deriving new predicates and adding them *locally* to the abstraction is only compatible with a boolean domain. – Our approach relies on weakest precondition-based refinement [9], which generally causes more overhead since new predicates are added *globally* to the abstraction. The question whether compatibility between three-valued abstraction and interpolation-based refinement can be established under certain conditions thus remains as future work.

A major advantage of our three-valued approach to abstraction is that it is capable of preserving both existential and universal properties of the considered system, whereas boolean abstractions are limited to either existential or universal preservation. The price that we pay is a generally higher complexity of the abstraction due to the enlarged domain for truth values. Thus, abstraction is always a trade-off between complexity and precision. Even our three-valued approach is in some cases unnecessarily imprecise: Verification may return *unknown* in cases where an additional look at the abstract model

would clearly reveal a definite result (compare section 4.2.3). More precision could be achieved by using the so-called *thorough* three-valued semantics for the evaluation of temporal logic properties [72], or by employing a four-valued abstraction [74]. However, both alternatives are connected with a significantly higher computational complexity. – In the future work section we will discuss the prospects of a three-valued abstraction that is complemented by quantified boolean variables, which allows for more precision without a significant increase of complexity.

Our heuristic approach to abstraction refinement is based on the evaluation of refinement candidates with regard to their benefit for the underlying verification task. The candidates are derived from unconfirmed counterexamples. In order to obtain a wider choice of potential refinement steps we use *multiple counterexample-generation*, which also enables us to eliminate more than one counterexample in each step. However, multiple counterexample-generation involves multiple explorations of the state space and thus additional computational overhead. Of course, a beneficial refinement candidate can compensate this overhead, but there is no guarantee that additional counterexamples yield better candidates. A possible alternative would be to expand the use of heuristics to counterexample-generation. *Heuristic-guided counterexample-generation* could be geared towards finding unconfirmed counterexamples that are particularly beneficial for abstraction refinement. In fact, the concept of using heuristics for the generation of counterexamples is not new. However, existing tools – in particular the model checker χChek [30, 54] that we employ in our framework – are tailored to generate *shortest* counterexamples, which are not necessarily useful for refinement. In the future work section we will present some ideas on how counterexample-generation for abstraction refinement can be enhanced by heuristics.

Finally, we want to turn our attention to the temporal logic properties that we considered in our approach. We decided to develop a verification framework for concurrent systems that supports full CTL model checking. CTL comprises several safety *and* liveness properties. – Liveness is of particular importance in concurrent systems where a number of processes compete for shared resources. Therefore, we integrated the existing multi-valued CTL model checker χChek into our framework. χChek can be regarded as a general-purpose model checker, since it supports full CTL and multi-valued domains of truth values. But the universality has its price. Specialised model checking tools that are restricted to safety and to a boolean domain typically exhibit a better performance. The mentioned restrictions prevent the direct integration of such tools into our framework. Nevertheless, there exist techniques for reducing three-valued model checking to boolean model checking [23, 126], and for reducing liveness checking to safety checking [20, 113]. Both reductions involve a significant growth of computational complexity. However, an investigation on whether these additional computational expenses might pay off in terms of facilitating the use of more efficient model checking tools is another direction for future research.

In conclusion, effective verification is always based on an appropriate compromise between different and potentially conflicting concepts. In our framework we decided for one possible combination of certain concepts, which gave us an encouraging approach to the verification of concurrent systems. A distinctive feature of our framework is the use of heuristics for abstraction refinement which essentially contributes to the success of our approach. Nevertheless, our discussion has revealed several opportunities for further improvements like the extension of the use of heuristics to counterexample-generation. Furthermore, we have discussed a number of alternatives to the concepts applied in our framework. These alternatives may outperform our current concepts in a particular way – however, at the price of additional expenses or restrictions. In the next section we will take a closer look at the prospects for enhancing our framework.

8.3 Future Work

The present work offers a number of interesting perspectives and topics for future research, which we will discuss in this section.

Heuristic-Guided Counterexample-Generation: The success of heuristic-guided abstraction refinement crucially depends on the choice of refinement candidates. Even a sophisticated decision procedure is of little value if the generated counterexample only hints at unfavourable candidates. Our current approach to this problem is *multiple counterexample-generation*, which we apply under the assumption that multiple counterexamples involve better choices for refinement. However, in the previous section we already discussed that this assumption may fail in several cases, and thus, multiple counterexample-generation may lead to an unnecessary computational overhead. We proposed *heuristic-guided counterexample-generation* as a promising alternative. In fact, research on this topic has already been started. Czech [48] developed a heuristic framework for the generation of unconfirmed counterexamples that are particularly beneficial for abstraction refinement. Criteria for the benefit of a counterexample are, among others, the probability of being refinable to a real counterexample and the costs for such a refinement. Both the probability and the costs are heuristically estimated based on a dependence analysis of the underlying system. Experiments on error detection in message passing systems revealed that the combination of heuristic-guided abstraction refinement and heuristic-guided counterexample-generation can lead to significant savings in the number of refinement steps. So far, the heuristic procedure for counterexample-generation has been prototypically implemented on top of model checker χChek. The applied heuristics are solely tailored to the *refutation* of temporal logic properties, i.e. only properties that turn out to be *false* can be checked more efficiently under these heuristics. Thus,

the development of heuristics that are specifically supportive for the *validation* of temporal logic properties is an expedient direction for future work. Moreover, our concept of heuristic-guided counterexample-generation for abstraction refinement could also initiate the development of an entirely new heuristic-based model checker.

Three-Valued Abstraction with Quantified Boolean Variables: Three-valued abstraction is one of the core concepts of our approach: Our verification framework constructs abstract models, or more precisely, *three-valued Kripke structures* of concurrent systems where transitions and labellings of states may take the additional truth value *unknown*. Temporal logic properties are evaluated on these Kripke structures with regard to the three-valued CTL semantics [29]. The corresponding model checking problem is PSPACE-complete. However, this approach is not maximally precise – an example: A property may definitely hold for all possible branches, but if one branching transition is *unknown* then the final outcome evaluates to *unknown* as well. We have already discussed that the required precision could be achieved by using the thorough temporal logic semantics [23], but at the price of EXPTIME-completeness. We believe that, due to the significantly higher complexity, the direct application of the thorough semantics is not an adequate alternative to our current approach. However, a slight modification of the general concept of the thorough semantics could be exploited for enhancing our framework. Model checking under the thorough semantics involves the construction of all completions of a three-valued Kripke structure. That is, all two-valued Kripke structures that can be obtained by substituting the *unknowns* with definite truth values are built. – Instead of such a global substitution, a small selection of *unknowns* could be replaced by *boolean variables*. These variables could then be universally quantified in a corresponding model checking problem, which would be equivalent to a *partial* use of the thorough semantics. Boolean variables in three-valued Kripke structures could also be exploited for modelling specific characteristics of the considered system, e.g. complementary branching transitions could be labelled with complementary variables. Hence, the combination of three-valued abstraction with quantified boolean variables could help to raise the precision at the relevant parts of an abstract model and thus to obtain more definite verification results without a significant increase of complexity. A similar approach has already been developed for the verification of hardware circuits [80]. Nevertheless, we expect that abstraction refinement for concurrent systems provides even more opportunities for effectively employing a mixed three-valued/quantified-boolean abstraction. Conceptual work on this topic has already begun [83].

Lighting Up the Shade: The basic idea of spotlight abstraction is to partition the processes of a concurrent system into a *spotlight* and a *shade*. Spotlight processes are thoroughly considered in the abstraction, whereas shade processes are nearly completely abstracted away by summarising them in one approximative shade component. This corresponds to a very rigorous approach to state space reduction with no scope for nuances of abstraction. Within this thesis, we already presented an extension of the original spotlight principle: *Shade clustering for message passing systems* introduces *multiple* shade components, or rather, shade clusters – one for each channel that is used for the communication between the shade processes and the spotlight. The extension permits us to preserve more definite behaviour of the shade processes at the price of a slightly larger abstraction. So far, shade clustering is based on a *fixed* cluster criterion (processes with common channels); it is tailored to a *one* field of application (message passing); and moreover, each cluster is reduced to a *single* control location associated with an approximating operation that is executed in a loop. – Hence, the concept of shade clustering offers several directions for future research. In order to preserve more behaviour with regard to a specific aspect of the system, a number of different clustering criteria are conceivable, for instance:

- processes that *share* a certain variable,
- processes that are *independent* from a certain variable,
- processes with a high degree of *similarity*,
  e.g. in terms of the number of shared variables.

Furthermore, the clustering criterion could be defined as a function of the current level of abstraction, i.e. depending on the current set of predicates and spotlight processes. In this way, the criterion could be dynamically revised during refinement. Another direction for future research concerns the abstraction of the shade clusters. Instead of reducing each cluster to just one control location, different degrees of summarisation could be employed (similar to our concept *region summarisation*). This would facilitate local adjustments of the degree of abstraction, e.g. tailored to the specific needs of the current verification task. Such a more differentiated approach to spotlight abstraction would also create new prospects for heuristic-guided refinement.

New Prospects for Heuristic-Guided Refinement: Our current procedure for heuristic-guided refinement basically consists of the following steps: We generate one or more unconfirmed counterexamples, derive a set of refinement candidates, heuristically select the best candidate and add it to the spotlight. Adding a *single* candidate in each iteration is a cautious

strategy for refinement which helps to avoid an unnecessary blow-up of the state space. However, this strategy generally leads to a large number of iterations and thus to a longer runtime. A promising way of improving refinement would be to introduce a heuristic evaluation function that determines the best *subset* of refinement candidates. Candidates that are tightly interrelated could be added as a set to the spotlight, for instance: a process together with a number of predicates that are relevant for characterising its behaviour, or a pair of complementary processes e.g. a producer together with a consumer. Such an approach could reduce the number of iterations and thus speed up verification. – In the previous two sections we proposed concepts for an enhanced abstraction of concurrent systems. These concepts also reveal new directions for heuristic-guided refinement: Unconfirmed counterexamples generally comprise transitions or predicates in states with the constant truth value *unknown*. Our current refinement procedure would try to resolve such uncertainty by adding a candidate to the spotlight. However, the concept of *three-valued abstraction with quantified boolean variables* would enable us to perform refinement in terms of replacing *unknown* constants by universally quantified variables – which could turn out to be the cheaper way to obtain the necessary precision in abstraction. Additional heuristic guidance could help to decide which way of refinement is currently the most expedient one. – Another new direction for refinement concerns the concept of introducing different degrees of abstraction for the shade: Instead of simply shifting process candidates from the abstraction stage *shade* to the stage *spotlight*, a number of intermediate stages could be used. This would give us more opportunities for balancing precision and complexity. Again, heuristics could be employed for determining the appropriate stage of abstraction for a process candidate.

Merging Counterexample-Generation and Refinement: So far, counterexample-generation and abstraction refinement are two separated steps in our framework: Unconfirmed counterexamples are always generated as complete paths from the initial state to some error state or cycle. In the next step, a refined abstract state space model is constructed and then counterexample-generation starts again from scratch. – Our proposed concept for *heuristic-guided counterexample-generation* could facilitate an *on-the-fly integration* of the steps counterexample-generation and refinement. Instead of always performing state space exploration from scratch, a confirmed prefix of an unconfirmed counterexample could be reused in the next iteration. The further expansion of such a prefix could be combined with abstraction refinement: Selecting a successor state for the prefix thus might involve the immediate addition of a certain predicate or process. Hence, refinement candidates would be no longer derived from entire counterexamples, but from the possible branches at the end of the confirmed prefix. The candidates could be heuristically evaluated as before. Additional

concepts for backtracking and discarding unfavourable counterexample prefixes could facilitate a revision of inexpedient exploration steps. Such a merged approach could significantly reduce the computational effort for counterexample-generation and abstraction refinement. A similar method has already been presented by Henzinger et al. in [79]. Their framework is based on boolean abstractions, verification is restricted to safety properties, and refinement is not performed under heuristic-guidance.

Combination with Complementary Reduction Techniques: Our approach to state space reduction is substantially based on the spotlight principle: Spotlight processes are considered in detail, whereas shade processes are nearly completely abstracted away. However, the spotlight itself exhibits the potential for additional reductions. *Partial order reduction* [64] is a method for cutting down the number of interleavings by exploiting the independence of concurrently executed operations. Hence, applying it to the parallel composition of processes in the spotlight could further reduce the complexity without causing any additional loss of precision. Moreover, *symmetry reduction* [58] can not only be used for reducing global verification tasks to local ones (compare section 6.2). Symmetry within the spotlight could be additionally exploited for summarising symmetric states. For parameterised verification tasks this would not involve any loss of precision.

References

1. Parosh Aziz Abdulla, Yu-Fang Chen, Giorgio Delzanno, Frederic Haziza, Chih-Duo Hong, and Ahmed Rezine. Constrained monotonic abstraction: A CEGAR for parameterized verification. In Paul Gastin and Francois Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, volume 6269 of *Lecture Notes in Computer Science*, pages 86–101. Springer-Verlag Berlin Heidelberg, 2010.
2. Parosh Aziz Abdulla, Giorgio Delzanno, Noomene Ben Henda, and Ahmed Rezine. Monotonic abstraction: On efficient verification of parameterized systems. *International Journal of Foundations of Computer Science*, 20(05):779–801, 2009.
3. Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Mayank Saksena. A survey of regular model checking. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 35–48. Springer-Verlag, 2004.
4. Luca Alfaro and Pritam Roy. Solving games via three-valued abstraction refinement. In Luis Caires and Vasco T. Vasconcelos, editors, *CONCUR 2007 - Concurrency Theory*, volume 4703 of *Lecture Notes in Computer Science*, pages 74–89. Springer-Verlag Berlin Heidelberg, 2007.
5. Frances E. Allen. Control flow analysis. *ACM SIGPLAN Notices - Proceedings of a Symposium on Compiler Optimization*, 5(7):1–19, July 1970.
6. Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.
7. Krzysztof R. Apt and Dexter Campbell Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6):307–309, May 1986.
8. Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.
9. Thomas Ball. Formalizing counterexample-driven refinement with weakest preconditions. In Manfred Broy, Johannes Grünbauer, David Harel, and Tony Hoare, editors, *Engineering Theories of Software Intensive Systems*, volume 195 of *NATO Science Series*, pages 121–139. Springer-Verlag Netherlands, 2005.
10. Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag Berlin Heidelberg, 2004.
11. Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 203–213, New York, NY, USA, 2001. ACM.
12. Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstraction for model checking C programs. In Tiziana Margaria and Wang Yi, editors,

Tools and Algorithms for the Construction and Analysis of Systems, volume 2031 of *Lecture Notes in Computer Science*, pages 268–283. Springer-Verlag Berlin Heidelberg, 2001.

13. Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Relative completeness of abstraction refinement for software model checking. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 158–172. Springer-Verlag Berlin Heidelberg, 2002.
14. Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In Klaus Havelund, John Penix, and Willem Visser, editors, *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130. Springer-Verlag Berlin Heidelberg, 2000.
15. Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 1–3, New York, NY, USA, 2002. ACM.
16. Kai Baukus, Yassine Lakhnech, and Karsten Stahl. Parameterized verification of a cache coherence protocol: Safety and liveness. In Agostino Cortesi, editor, *Verification, Model Checking, and Abstract Interpretation*, volume 2294 of *Lecture Notes in Computer Science*, pages 317–330. Springer-Verlag Berlin Heidelberg, 2002.
17. Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering, 2007. FOSE '07*, pages 85–103, 2007.
18. Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast: Applications to software engineering. *Int. J. Softw. Tools Technol. Transf.*, 9(5):505–525, October 2007.
19. Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer-Verlag Berlin Heidelberg, 2011.
20. Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. *Electronic Notes in Theoretical Computer Science*, 66(2):160 – 177, 2002.
21. Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag Berlin Heidelberg, 1999.
22. Glenn Bruns and Patrice Godefroid. Model checking partial state spaces with 3-valued temporal logics. In Nicolas Halbwachs and Doron Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, pages 274–287, London, UK, 1999. Springer-Verlag Berlin Heidelberg.
23. Glenn Bruns and Patrice Godefroid. Generalized model checking: Reasoning about partial state spaces. In Catuscia Palamidessi, editor, *CONCUR 2000 - Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, pages 168–182. Springer-Verlag Berlin Heidelberg, 2000.
24. Glenn Bruns and Patrice Godefroid. Model checking with multi-valued logics. In Josep Diaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella, editors, *Automata, Languages and Programming*, volume 3142 of *Lecture Notes in Computer Science*, pages 281–293. Springer-Verlag Berlin Heidelberg, 2004.
25. Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.
26. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science, 1990. LICS '90*, pages 428–439, 1990.

27. Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, June 2004.
28. Marsha Chechik, Benet Devereux, Steve M. Easterbrook, and Arie Gurfinkel. Multi-valued symbolic model-checking. *ACM Transactions on Software Engineering and Methodology*, 12(4):371–408, 2003.
29. Marsha Chechik, Steve Easterbrook, and Victor Petrovykh. Model-checking over multi-valued logics. In Jose Nuno Oliveira and Pamela Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity*, volume 2021 of *Lecture Notes in Computer Science*, pages 72–98. Springer-Verlag Berlin Heidelberg, 2001.
30. Marsha Chechik, Arie Gurfinkel, and Benet Devereux. χChek: A multi-valued model-checker. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 505–509. Springer-Verlag Berlin Heidelberg, 2002.
31. Zbenqiang Chen, Baowen Xu, Hongji Yang, Kecheng Liu, and Jianping Zhang. An approach to analyzing dependency of concurrent programs. In *Proceedings of the First Asia-Pacific Conference on Quality Software, 2000*, pages 39–43, 2000.
32. Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.
33. Alessandro Cimatti, Enrico Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Integrating bdd-based and sat-based symbolic model checking. In Alessandro Armando, editor, *Frontiers of Combining Systems*, volume 2309 of *Lecture Notes in Computer Science*, pages 49–56. Springer-Verlag Berlin Heidelberg, 2002.
34. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and Aravinda Prasad Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer-Verlag Berlin Heidelberg, 2000.
35. Edmund Clarke, Muralidhar Talupur, and Helmut Veith. Environment abstraction for parameterized verification. In E. Allen Emerson and Kedar S. Namjoshi, editors, *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 126–141. Springer-Verlag Berlin Heidelberg, 2006.
36. Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag Berlin Heidelberg, 1982.
37. Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, April 1986.
38. Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 343–354, New York, NY, USA, 1992. ACM.
39. Edmund M. Clarke, Somesh Jha, Reinhard Enders, and Thomas Filkorn. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1-2):77–104, 1996.
40. Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, December 1996.
41. Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.
42. Jamieson M. Cobleigh, Lori A. Clarke, and Leon J. Osterweil. FLAVERS: A finite state verification technique for software systems. *IBM Systems Journal*, 41(1):140–165, 2002.

43. Mika Cohen, Mads Dam, Alessio Lomuscio, and Hongyang Qu. A data symmetry reduction technique for temporal-epistemic logic. In Zhiming Liu and Anders P. Ravn, editors, *Automated Technology for Verification and Analysis*, volume 5799 of *Lecture Notes in Computer Science*, pages 69–83. Springer-Verlag Berlin Heidelberg, 2009.
44. C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In Edmund M. Clarke and Robert P. Kurshan, editors, *Computer-Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 233–242. Springer-Verlag Berlin Heidelberg, 1991.
45. P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with readers and writers. *Communications of the ACM*, 14(10):667–668, October 1971.
46. Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.
47. Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 269–282, New York, NY, USA, 1979. ACM.
48. Mike Czech. *Dreiwertiges Model Checking paralleler Systeme mit heuristisch geleiteter Generierung von Gegenbeispielen*. Bachelor's thesis, University of Paderborn, February 2013.
49. Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.
50. Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.
51. Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, October 1972.
52. Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.
53. Alastair F. Donaldson, Alexander Kaiser, Daniel Kroening, Michael Tautschnig, and Thomas Wahl. Counterexample-guided abstraction refinement for symmetric concurrent programs. *Formal Methods in System Design*, 41(1):25–44, 2012.
54. Steve M. Easterbrook, Marsha Chechik, Benet Devereux, Arie Gurfinkel, Albert Y. C. Lai, Victor Petrovykh, Anya Tafliovich, and Christopher Thompson-Walsh. χChek: A model checker for multi-valued reasoning. In *Proceedings of the 25th International Conference on Software Engineering, 2003*, pages 804–805, 2003.
55. Stefan Edelkamp and Shahid Jabbar. Large-scale directed model checking LTL. In Antti Valmari, editor, *Model Checking Software*, volume 3925 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag Berlin Heidelberg, 2006.
56. Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue. Directed explicit model checking with HSF-SPIN. In Matthew Dwyer, editor, *Model Checking Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 57–79. Springer-Verlag Berlin Heidelberg, 2001.
57. E. Allen Emerson and Vineet Kahlon. Reducing model checking of the many to the few. In David McAllester, editor, *Automated Deduction - CADE-17*, volume 1831 of *Lecture Notes in Computer Science*, pages 236–254. Springer-Verlag Berlin Heidelberg, 2000.
58. E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. In Costas Courcoubetis, editor, *Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 463–478. Springer-Verlag Berlin Heidelberg, 1993.
59. E. Allen Emerson and Thomas Wahl. On combining symmetry reduction and symbolic representation for efficient model checking. In Daniel Geist and Enrico Tronci, editors, *Correct Hardware Design and Verification Methods*, volume 2860 of *Lecture Notes in Computer Science*, pages 216–230. Springer-Verlag Berlin Heidelberg, 2003.
60. Javier Esparza, Stefan Kiefer, and Stefan Schwoon. Abstraction refinement with craig interpolation and symbolic pushdown systems. In Holger Hermanns and Jens

Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 489–503. Springer-Verlag Berlin Heidelberg, 2006.

61. Harald Fecher and Sharon Shoham. Local abstraction-refinement for the μ-calculus. In Dragan Bosnacki and Stefan Edelkamp, editors, *Proceedings of the 14th International SPIN Conference on Model checking Software*, volume 4595 of *Lecture Notes in Computer Science*, pages 4–23. Springer-Verlag Berlin Heidelberg, 2007.
62. Melvin Fitting. Kleene's three valued logics and their children. *Fundamenta Informaticae*, 20(1-3):113–131, March 1994.
63. Marcelo Glusman, Gila Kamhi, Sela Mador-Haim, Ranan Fraer, and Moshe Y. Vardi. Multiple-counterexample guided iterative abstraction refinement: An industrial evaluation. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 176–191. Springer-Verlag Berlin Heidelberg, 2003.
64. Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.
65. Patrice Godefroid and Radha Jagadeesan. Automatic abstraction using generalized model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceedings of the 14th International Conference on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 137–151. Springer-Verlag Berlin Heidelberg, 2002.
66. Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer-Verlag Berlin Heidelberg, 1997.
67. Orna Grumberg. 2-valued and 3-valued abstraction-refinement in model checking. In *Logics and Languages for Reliability and Security*, pages 105–128. IOS Press, Incorporated, 2010.
68. Orna Grumberg, Martin Lange, Martin Leucker, and Sharon Shoham. *Don't Know* in the μ-calculus. In Radhia Cousot, editor, *Proceedings of the 6th international conference on Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *Lecture Notes in Computer Science*, pages 233–249. Springer-Verlag Berlin Heidelberg, 2005.
69. Orna Grumberg, Martin Lange, Martin Leucker, and Sharon Shoham. When not losing is better than winning: Abstraction and refinement for the full μ-calculus. *Information and Computation*, 205(8):1130 – 1148, 2007.
70. Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 331–344, New York, NY, USA, 2011. ACM.
71. Arie Gurfinkel and Marsha Chechik. Multi-valued model checking via classical model checking. In Roberto Amadio and Denis Lugiez, editors, *CONCUR 2003 - Concurrency Theory*, volume 2761 of *Lecture Notes in Computer Science*, pages 266–280. Springer-Verlag Berlin Heidelberg, 2003.
72. Arie Gurfinkel and Marsha Chechik. How thorough is thorough enough? In Dominique Borrione and Wolfgang Paul, editors, *Correct Hardware Design and Verification Methods*, volume 3725 of *Lecture Notes in Computer Science*, pages 65–80. Springer-Verlag Berlin Heidelberg, 2005.
73. Arie Gurfinkel and Marsha Chechik. Why waste a perfectly good abstraction? In Holger Hermanns and Jens Palsberg, editors, *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 212–226. Springer-Verlag Berlin Heidelberg, 2006.

74. Scott Hazelhurst and Carl-Johan H. Seger. Model checking lattices: Using and reasoning about information orders for abstraction. *Logic Journal of the IGPL*, 7(3):375–411, 1999.
75. Fei He, Xiaoyu Song, Ming Gu, and Jiaguang Sun. Effective heuristics for counterexample-guided abstraction refinement. In *Proceedings of the 17th ACM Great Lakes Symposium on VLSI*, GLSVLSI '07, pages 393–398, New York, NY, USA, 2007. ACM.
76. Fei He, Xiaoyu Song, Ming Gu, and Jiaguang Sun. Heuristic-guided abstraction refinement. *The Computer Journal*, 52(3):280–287, May 2009.
77. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 232–244, New York, NY, USA, 2004. ACM.
78. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. Thread-modular abstraction refinement. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 262–274. Springer-Verlag Berlin Heidelberg, 2003.
79. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 58–70, New York, NY, USA, 2002. ACM.
80. Marc Herbstritt and Bernd Becker. On combining 01X-logic and QBF. In Roberto Moreno Diaz, Franz Pichler, and Alexis Quesada Arencibia, editors, *Computer Aided Systems Theory - EUROCAST 2007*, volume 4739 of *Lecture Notes in Computer Science*, pages 531–538. Springer-Verlag Berlin Heidelberg, 2007.
81. Jörg Hoffmann, Jan-Georg Smaus, Andrey Rybalchenko, Sebastian Kupferschmid, and Andreas Podelski. Using predicate abstraction to generate heuristic functions in UPPAAL. In Stefan Edelkamp and Alessio Lomuscio, editors, *Model Checking and Artificial Intelligence*, volume 4428 of *Lecture Notes in Computer Science*, pages 51–66. Springer-Verlag Berlin Heidelberg, 2007.
82. G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
83. Nikolaos Ikonomakis. *Combining Three-Valued Logic and Quantified Boolean Formulae in Bounded Model Checking Encodings*. Master's thesis, University of Paderborn, February 2013.
84. Radu Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ASE '01, pages 254–261, Washington, DC, USA, 2001. IEEE Computer Society.
85. Cliff B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP Congress*, pages 321–332, 1983.
86. Alexander Kaiser, Daniel Kroening, and Thomas Wahl. Dynamic cutoff detection in parameterized concurrent programs. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 645–659. Springer-Verlag Berlin Heidelberg, 2010.
87. Joost-Pieter Katoen, Daniel Klink, Martin Leucker, and Verena Wolf. Three-valued abstraction for probabilistic systems. *The Journal of Logic and Algebraic Programming*, 81(4):356 – 389, 2012.
88. Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1978.
89. Ekkart Kindler. Safety and liveness properties: a survey. *Bulletin of the European Association for Theoretical Computer Science*, 53:268–272, 1994.
90. Saul Kripke. Semantical considerations on modal logic. *Acta Phil. Fennica*, 16:83–94, 1963.
91. Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, Princeton, NJ, USA, 1994.

92. Marta Z. Kwiatkowska. Survey of fairness notions. *Inf. Softw. Technol.*, 31(7):371–386, September 1989.
93. Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.
94. Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.
95. Kenneth L. McMillan. Interpolation and SAT-based model checking. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag Berlin Heidelberg, 2003.
96. Kenneth L. McMillan. Applications of Craig interpolation to model checking. In Gianfranco Ciardo and Philippe Darondeau, editors, *Applications and Theory of Petri Nets 2005*, volume 3536 of *Lecture Notes in Computer Science*, pages 15–16. Springer-Verlag Berlin Heidelberg, 2005.
97. Kenneth L. McMillan, Shaz Qadeer, and James B. Saxe. Induction in compositional model checking. In E. Allen Emerson and Aravinda Prasad Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 312–327. Springer-Verlag Berlin Heidelberg, 2000.
98. Kennethn L. McMillan. Lazy abstraction with interpolants. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer-Verlag Berlin Heidelberg, 2006.
99. Yael Meller, Orna Grumberg, and Sharon Shoham. A framework for compositional verification of multi-valued systems via abstraction-refinement. In Zhiming Liu and Anders P. Ravn, editors, *Automated Technology for Verification and Analysis*, volume 5799 of *Lecture Notes in Computer Science*, pages 271–288. Springer-Verlag Berlin Heidelberg, 2009.
100. Björn Metzler, Heike Wehrheim, and Daniel Wonisch. Decomposition for compositional verification. In Shaoying Liu, Tom Maibaum, and Keijiro Araki, editors, *Formal Methods and Software Engineering*, volume 5256 of *Lecture Notes in Computer Science*, pages 105–125. Springer-Verlag Berlin Heidelberg, 2008.
101. Alice Miller, Alastair Donaldson, and Muffy Calder. Symmetry in temporal logic model checking. *ACM Comput. Surv.*, 38(3), September 2006.
102. Leonardo Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer-Verlag Berlin Heidelberg, 2008.
103. Kedar S. Namjoshi. Symmetry and completeness in the analysis of parameterized systems. In Byron Cook and Andreas Podelski, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 4349 of *Lecture Notes in Computer Science*, pages 299–313. Springer-Verlag Berlin Heidelberg, 2007.
104. Gleb Naumovich, Lori A. Clarke, Leon J. Osterweil, and Matthew B. Dwyer. Verification of concurrent software with FLAVERS. In *Proceedings of the 19th International Conference on Software Engineering*, ICSE '97, pages 594–595, New York, NY, USA, 1997. ACM.
105. C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1-2):41–75, 1996.
106. Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976.
107. Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.
108. Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.
109. Amir Pnueli, Sitvanit Ruah, and Lenore Zuck. Automatic deductive verification with invisible invariants. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms*

for the Construction and Analysis of Systems, volume 2031 of *Lecture Notes in Computer Science*, pages 82–97. Springer-Verlag Berlin Heidelberg, 2001.

110. Amir Pnueli, Jessie Xu, and Lenore Zuck. Liveness with (0,1, ∞)-counter abstraction. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 107–122. Springer-Verlag Berlin Heidelberg, 2002.
111. Xiaofang Qi and Baowen Xu. Dependence analysis of concurrent programs based on reachability graph and its applications. In Marian Bubak, GeertDick Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *Computational Science - ICCS 2004*, volume 3036 of *Lecture Notes in Computer Science*, pages 405–408. Springer-Verlag Berlin Heidelberg, 2004.
112. Jonas Schrieb, Heike Wehrheim, and Daniel Wonisch. Three-valued spotlight abstractions. In Ana Cavalcanti and Dennis R. Dams, editors, *FM 2009: Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 106–122. Springer-Verlag Berlin Heidelberg, 2009.
113. Viktor Schuppan and Armin Biere. Liveness checking as safety checking for infinite state spaces. *Electronic Notes in Theoretical Computer Science*, 149(1):79 – 96, 2006.
114. Sharon Shoham. *Abstraction-Refinement and Modularity in μ-Calculus Model Checking*. Phd thesis, Department of Computer Science, Technion - Israel Institute of Technology, 2009.
115. Sharon Shoham and Orna Grumberg. 3-valued abstraction: More precision at less cost. *Information and Computation*, 206(11):1313 – 1333, 2008.
116. A. Srinivasan, T. Ham, S. Malik, and R.K. Brayton. Algorithms for discrete function manipulation. In *IEEE International Conference on Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers*, pages 92–95, 1990.
117. Boleslaw K. Szymanski. A simple solution to Lamport's concurrent programming problem with linear wait. In *Proceedings of the 2nd International Conference on Supercomputing*, ICS '88, pages 621–626, New York, NY, USA, 1988. ACM.
118. Jianbin Tan, George S. Avrunin, and Lori A. Clarke. Heuristic-based model refinement for FLAVERS. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 635–644, 2004.
119. Nils Timm and Heike Wehrheim. On symmetries and spotlights – verifying parameterised systems. In JinSong Dong and Huibiao Zhu, editors, *Formal Methods and Software Engineering*, volume 6447 of *Lecture Notes in Computer Science*, pages 534–548. Springer-Verlag Berlin Heidelberg, 2010.
120. Nils Timm, Heike Wehrheim, and Mike Czech. Heuristic-guided abstraction refinement for concurrent systems. In Toshiaki Aoki and Kenji Taguchi, editors, *Formal Methods and Software Engineering*, volume 7635 of *Lecture Notes in Computer Science*, pages 348–363. Springer-Verlag Berlin Heidelberg, 2012.
121. Tobe Toben. Counterexample guided spotlight abstraction refinement. In Kenji Suzuki, Teruo Higashino, Keiichi Yasumoto, and Khaled El-Fakih, editors, *Proceedings of the 28th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems - FORTE 2008*, volume 5048 of *Lecture Notes in Computer Science*, pages 21–36. Springer-Verlag Berlin Heidelberg, 2008.
122. J. Hooiman Y. Lakhneche M. Poel J. Zwiers F. de Boer. W. de Roever, U. Hanneman. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, New York, NY, USA, 2001.
123. Björn Wachter and Bernd Westphal. The spotlight principle: on combining process-summarizing state abstractions. In Byron Cook and Andreas Podelski, editors, *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 4349 of *Lecture Notes in Computer Science*, pages 182–198. Springer-Verlag Berlin Heidelberg, 2007.
124. Thomas Wahl, Nicolas Blanc, and E. Allen Emerson. SVISS: symbolic verification of symmetric systems. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and*

Algorithms for the Construction and Analysis of Systems, volume 4963 of *Lecture Notes in Computer Science*, pages 459–462. Springer-Verlag Berlin Heidelberg, 2008.

125. Chao Wang, Hyondeuk Kim, and Aarti Gupta. Hybrid CEGAR: combining variable hiding and predicate abstraction. In *Proceedings of the 2007 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '07, pages 310–317, Piscataway, NJ, USA, 2007. IEEE Press.
126. Heike Wehrheim. Bounded model checking for partial Kripke structures. In John S. Fitzgerald, Anne E. Haxthausen, and Husnu Yenigun, editors, *Theoretical Aspects of Computing - ICTAC 2008*, volume 5160 of *Lecture Notes in Computer Science*, pages 380–394. Springer-Verlag Berlin Heidelberg, 2008.
127. Daniel Wonisch. Block abstraction memoization for CPAchecker. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7214 of *Lecture Notes in Computer Science*, pages 531–533. Springer-Verlag Berlin Heidelberg, 2012.